Implementing and Evaluating the DYMO Routing Protocol

Master's Thesis

ROLF EHRENREICH THORUP

ADVISOR: LARS KRISTENSEN

FEBRUARY, 2007

DEPARTMENT OF COMPUTER SCIENCE
UNIVERSITY OF AARHUS
DENMARK

Abstract

In a Mobile Ad hoc Network (MANET), mobile nodes move around arbitrarily, nodes may join and leave at any time, and the resulting topology is constantly changing. Routing in a MANET is challenging because of the dynamic topology and the lack of an existing fixed infrastructure. The Dynamic MANET On-demand (DYMO) protocol builds on previous proposed routing protocols for MANETs. This thesis present a design and implementation of DYMO for Linux and an experimental practical evaluation of DYMO with respect to quantitative performance metrics. The thesis contains a survey of implementation challenges that stem from the lack of system-service support for on-demand MANET routing protocols and a survey of implementation solutions addressing these challenges. The actual implementation consists of a user space routing daemon and a Linux kernel module, based on the netfilter framework. In the practical evaluation, the measured metrics are route discovery latency, TCP and UDP throughput, and end-to-end latency. Many of the experiments have been conducted in both an emulated and a real setup. Furthermore, the thesis present measurements to determine the limits of TCP based services in MANETs, the so-called ad hoc horizon. The results of the experiments showed performance in the emulated setup to be better than the real setup. In the UDP and TCP experiments, the maximum achieved throughput was 3-4 times better in the emulated setup than in the real setup. Furthermore, the evaluation showed results to be comparable to previous experiments with the related Ad hoc On-demand Distance Vector protocol. The experiments involving the ad hoc horizon indicated that conclusions based on previously obtained simulation results are correct.

Danish Summary

Et *mobilt ad hoc-netværk* (MANET) består af en mængde af mobile enheder som bevæger sig frit rundt, og som selv danner en rutningsinfrastruktur, så hver enhed fungerer som router. Rutning i et mobilt ad hoc-netværk er en stor udfordring på grund af den dynamiske typologi og typiske hardware-variation for eksempel forskel i radiosendestyrke og batterilevetid. *Dynamic MANET On-demand* (DYMO) rutningsprotokollen bygger på tidligere erfaringer med eksisterende protokoller bl.a. *Ad hoc On-demand Distance Vector* (AODV) og *Distance Vector Routing* protokollerne og er på nuværende tidspunkt centrum i det arbejde, som bliver udført af MANET arbejdsgruppen under *the Internet Engineering Task Force*. Målet for dette speciale er at lave et review af DYMO specifikationen ved at implementere denne, samt at lave en evaluering af protokollen ved brug af praktiske eksperimenter.

Implementation

Specialet præsenterer design og implementation af DYMO til Linux. For at kunne implementere DYMO undersøges de udfordringer der ligger i at implementere en on-demand MANET protokol. Udfordringerne er en konsekvens af nuværende operativsystemers manglende understøttelse til at identificere de hændelser, som er essentielle for en on-demand protokol. Herefter præsenterer specialet en række løsningsmuligheder og designvalg, som kan benyttes til at implementere DYMO til Linux. De nævnte løsningsmetoder er kernekode modifikation, snooping samt netfilter hooks. Den faktiske implementation består af en user-space rutningsdæmon (*eng: routing daemon*) og et Linux kernemodul og benytter netfilter. Implementationen af rutningsdæmonen er skrevet i sprogene *Lua* og *C*. Selve rutningslogikken er skrevet i Lua, og denne del er indlejret i en C-del, som implementerer netværks-I/O og main loop. Kernemodulet er skrevet i C og implementerer netfilter-delen, håndtering af aktive ruter, pakkekø samt kommunikation med rutningsdæmonen i user-space. Vi har i implementationen af rutningsdæmonen fokuseret på porterbarhed. Derudover giver vi en beskrivelse af, hvordan mængden af trafik mellem kernelmodul og rutningsdæmon kan begrænses samtidig med at rutningsdæmonen holdes tilstrækkelig opdateret.

Eksperimenter

Valget af de udførte eksperimenter er baseret på en gennemgang af tidligere eksperimenter typisk udført med protokollen AODV. Dette skyldes at der i skrivende stund ikke er lavet praktiske evalueringer af DYMO, men kun evalueringer baseret på simulation. De udførte og beskrevne eksperimenter er route discovery latency,

UDP og TCP throughput samt end-to-end delay. Derudover er der udført eksperimenter med formål at bestemme grænser for brugbarhed af services baseret på TCP i ad hoc-netværk. Denne grænse benævnes ad hoc horisonten, og er et mål for det maksimale antal af hop eller enheder, der kan være i et netværk før svartider og hastighed føles utilstrækkelig. De fleste eksperimenter er udført i to forskellige opstillinger. Dels i en (emuleret) opstilling hvor bærbare computere er placeret i det samme lokale, men hvor forbindelserne mellem enheder programmatisk er brudt ifølge den ønskede netværkstopologi. Dels i en opstilling hvor de bærbare computere er placeret fysisk adskilt i overensstemmelse med den ønskede topologi. De overordnede resultater opnået i forbindelse med eksperimenterne viser, at resultaterne i den emulerede opstilling er bedre end i den virkelige opstilling. I målingerne af UDP og TCP hastighed er throughput 3-4 gange bedre i den emulerede opstilling. Målingerne i forbindelse med ad hoc horisonten bekræfter tidligere simulationsresultater i at ruter på og over 3 hop medfører forringet anvendelighed af services baseret på TCP.

Acknowledgements

I thank my advisor Lars Kristensen for his excellent support and guidance during the course of this project, for supplying me with hardware and additionally extending the period I could have access to the hardware. The support I have received has been indispensable and unsurpassed.

I thank Jeppe Brønsted for useful suggestions and comments. I am grateful to Peter Gade Jensen for helping with office discipline during parts of the writing process and for the countless pots of coffee shared. In addition, I am grateful for his and Thomas Jakobsen's company on numerous Fridays in Ada-020.

I thank my mother Clara Ehrenreich for proof-reading the thesis draft and I thank all my family for their patience and encouragement. Finally, a special thanks goes to Anna Laura for her understanding and for keeping me sane throughout.

Contents

1 Introduction **1**
- 1.1 Mobile Ad Hoc Networks 1
- 1.2 The DYMO Routing Protocol 2
- 1.3 Evaluation of MANET Routing Protocols 3
  - 1.3.1 Evaluation of MANET Routing Protocols using Network Simulators 3
  - 1.3.2 Experimental Evaluation of the DYMO Routing Protocol . 3
- 1.4 Aim of Thesis 4
- 1.5 Methods 5
- 1.6 Thesis Outline 5

2 Mobile Ad Hoc Networks and Routing Protocols **7**
- 2.1 Application of MANETs 8
- 2.2 Conventional Wired Network Routing Protocols 8
  - 2.2.1 Distance Vector Routing 9
  - 2.2.2 Link State Routing 10
- 2.3 MANET Routing protocols 11
  - 2.3.1 On-Demand Routing Protocols 11
  - 2.3.2 Table-Driven Routing Protocols 11
  - 2.3.3 MANET Routing Protocol Challenges 12
- 2.4 The AODV Routing Protocol 12
  - 2.4.1 Route Discovery 13
  - 2.4.2 Route Maintenance 14
- 2.5 The DSR Protocol 15
  - 2.5.1 Basic Route Discovery 15
  - 2.5.2 Route Maintenance 16
  - 2.5.3 Route Discovery Optimizations 17
  - 2.5.4 Route Maintenance Optimizations 17
- 2.6 The OLSR Protocol 18
  - 2.6.1 Neighbour Discovery 20

3 The DYMO Routing Protocol **21**
- 3.1 Protocol Overview 22
- 3.2 Route Discovery 23
- 3.3 Route Maintenance 25
- 3.4 Generalized Packet and Message Format 27
  - 3.4.1 The Message Header 27
  - 3.4.2 The Message Body 29

4 Implementation Approach **31**
4.1 Challenges . . . 32
4.1.1 Identifying the On-demand Ad Hoc Routing Challenges . 33
4.2 Implementation Techniques on Linux . . . 34
4.2.1 Kernel Modification . . . 36
4.2.2 Snooping . . . 36
4.2.3 Netfilter . . . 38
4.2.4 Additional Implementation Issues . . . 40

5 DYMO-AU Design and Implementation Overview **43**
5.1 Design Approach . . . 43
5.2 Implementation Overview . . . 44
5.2.1 Packet Queue . . . 45
5.2.2 Expiry List . . . 45
5.2.3 Netlink Communication . . . 45
5.2.4 Netfilter Hooks . . . 45
5.3 The Lua Programming Language . . . 46
5.4 User Space-Kernel Space Interaction . . . 47
5.4.1 Message Types . . . 47
5.4.2 Communication Interface in the Daemon . . . 48
5.4.3 Route Discovery Example . . . 48
5.4.4 RERR Processing Example . . . 49
5.5 Discussion . . . 49
5.5.1 Errors in the DYMO Specification . . . 50
5.5.2 Limitations of the DYMO-AU implementation . . . 51
5.5.3 Portability . . . 51

6 DYMO-AU Design and Implementation Details **53**
6.1 The User Space Routing Daemon . . . 53
6.1.1 Timer Queue . . . 53
6.1.2 `select` I/O Multiplexing Main Loop . . . 55
6.1.3 DYMO and Control Packet Dispatching . . . 56
6.1.4 DYMO Message Processing . . . 57
6.1.5 Routing Message Processing . . . 57
6.1.6 Route Discovery . . . 58
6.1.7 Route Error Processing . . . 58
6.1.8 Routing Table . . . 58
6.2 The Kernel Module . . . 61
6.2.1 Packet Queue . . . 61
6.2.2 Expiry List . . . 62
6.2.3 Netlink Communication . . . 63
6.2.4 Netfilter Hooks . . . 64
6.3 Updating Route Timeouts . . . 66
6.3.1 Packet-triggered Update of Timeouts . . . 66

6.3.2 Timeout-triggered Update of Timeouts 68
6.3.3 On-demand Update of Timeouts 70

7 Experimental Evaluation 73
7.1 Related Work and Testbeds 74
7.1.1 Evaluation Testbeds 76
7.1.2 Summary 79
7.2 Experiments 79
7.2.1 Experimental Set Up 81
7.3 Route Discovery Latency 82
7.3.1 MobiEmu Setup 83
7.3.2 Real Setup 84
7.3.3 Comparing MobiEmu and Real Setup Results 85
7.4 UDP Performance 86
7.5 End-to-End Delay 89
7.6 TCP Performance 92
7.6.1 FTP Performance 93
7.7 Ad Hoc Horizon 95
7.7.1 Measuring TCP Unfairness 96
7.7.2 Measuring HTTP Download Times 97
7.8 Experiences Learned 99

8 Conclusions and Future Work 101
8.1 Summary 101
8.1.1 Implementation 101
8.1.2 Experimental Evaluation 103
8.2 Conclusions 104
8.2.1 Implementation 104
8.2.2 Experimental Evaluation 105
8.3 Future Work and Research 105
8.3.1 Implementation 105
8.3.2 Practical Evaluation 108

A Setting Linux Kernel Parameters 109

B Contents of the CD-ROM 111

References 113

1 Introduction

The use of wireless technology has become a ubiquitous method to access the Internet or connect to the local network whether in a corporate, educational, or private setting. Practically all laptop computers are currently sold with a built-in wireless adapter. In handheld units like PDAs, wireless adapters have also become standard and are now being introduced in some types of mobile phones. It is much easier and inexpensive to deploy a wireless network compared to a traditional wired network, as the required effort and cost of running cables are negligible. Furthermore, additional devices can be added to the network at no extra cost.

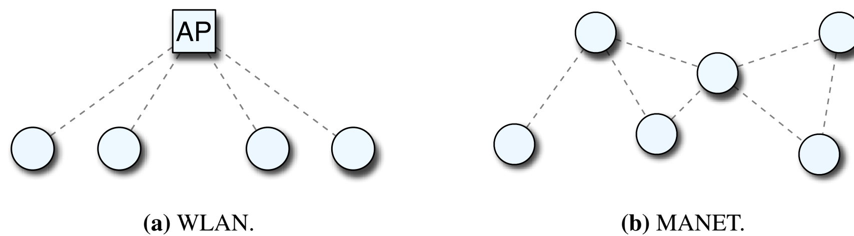
In order for a wireless equipped device to access other computers on the (wireless) local network or connect to the Internet it must associate with a wireless access point. A wireless access point is a device that allows devices equipped with wireless adapters to be linked together in a local area network (LAN) and to connect to a preexisting wired LAN and via a gateway to get access to the Internet. Such networks are called wireless local area networks (WLANs) as the wireless access point is linking wireless devices without wires. Because of the convenience of not having to rely on wires, WLANs have become immensely popular.

When devices equipped with wireless adapters are part of a WLAN and are managed by a wireless access point, their coordination is controlled by a centralized entity. The devices rely on the presence of a fixed infrastructure, i.e., wireless access points to work. Laptop computers must be within the range of a wireless access point to connect to other devices because the laptops must communicate via the access point.

1.1 Mobile Ad Hoc Networks

If communication between wireless equipped devices is desired, the reliance upon an existing infrastructure as well as its implied limitations on mobility can be a major obstacle. In such cases, the wireless equipped devices themselves must operate autonomously to provide connection such that a device not directly within transmission range of another device is able to communicate. Each wireless capable

device must function as a router and forward packets. Thus, communication can be via multiple wireless hops. In the following, such wireless equipped devices are referred to as *nodes*. Additional challenges arise as nodes may move around arbitrarily resulting in networks with constantly changing, random multi-hop topologies. Such a network is called a *mobile ad hoc network* (MANET) because the nodes in the network are mobile and communicate without a pre-established fixed infrastructure, but instead form a routing infrastructure in an ad hoc fashion. Figure 1.1 shows the difference between a WLAN and a MANET. In a WLAN, the mobile nodes are managed by the wireless access point. In a MANET, the mobile nodes must work together in a distributed fashion to enable routing among the nodes. Because of the lack of centralized control, routing becomes a central issue and a major challenge as the network topology is constantly changing. The mobility patterns and the condition under which a routing protocol is supposed to work can vary considerably. Furthermore, the number of mobile nodes in the network can range from a few to several hundreds or thousands. Because of the diverse envisioned working conditions, several MANET routing protocols have been proposed.



(a) WLAN. **(b)** MANET.

Figure 1.1: Nodes in a WLAN managed by an access point and nodes in MANET independently forming a routing infrastructure.

1.2 The DYMO Routing Protocol

An example of a routing protocol for MANETs is the *Dynamic MANET On-demand* (DYMO) routing protocol [CP06a]. The DYMO routing protocol is a recently proposed protocol currently defined in an IETF Internet-Draft and is thus, work in progress. It is currently in its sixth version. DYMO belongs to the category of MANET routing protocols called *on-demand* or *reactive* routing protocols. An on-demand protocol only tries to discover a route to a destination, when it is actually needed by an application.

The DYMO protocol and the specification of it are new, and at the time of writing two implementations exist, NIST DYMO [KBb] and DYMOUM [RR]. To evaluate a protocol specification, especially a protocol draft, it is important that several implementations are made available by independent sources. An implementation is a review of the specification as it is necessary to carefully read and

understand the specification in order to implement it. In addition, when several implementations are available they can be tested for interoperability. If two implementations are found not to be interoperable, it can be because the specification is unclear and parts of it can be interpreted wrongly. Eventually, for an Internet-Draft to be promoted to an RFC at least two independent implementations must exist and be interoperable [Bra96].

1.3 Evaluation of MANET Routing Protocols

One of the ways we can compare different MANET routing protocols is to compare quantitative performance metrics measured while conducting experimental evaluations in a MANET. Examples of such metrics include end-to-end delay, route discovery latency, and average number of bits transmitted compared to the number of bits delivered.

1.3.1 Evaluation of MANET Routing Protocols using Network Simulators

One way to perform such measurements involving MANET routing protocols is with the use of network simulators. *Ns-2* [BEF+00] is an example of a simulator that is heavily used in MANET research. A network simulator is a valuable tool to allow experiments and evaluations of protocols in an environment that is easy to control and where changing the test parameters, for instance, the topology, is easy. Accordingly, a lot of research in MANET routing protocols involving protocol evaluation is conducted with the use of network simulators. However, a simulator inevitably leave out some of the characteristics of a real network, as the real world behaviour of radio waves propagation is complex and thus difficult and computational complex to mimic in a simulator [KNG+04]. Using a network simulator is a convenient way to assist with the evaluation of MANET routing protocols, but it is also important to test implementations of MANET protocols outside a simulator, as a protocol must be tested in the real world environments in which it was envisioned to work.

1.3.2 Experimental Evaluation of the DYMO Routing Protocol

An experimental evaluation of the DYMO routing protocol is interesting as the DYMO specification is new and few results have previously been obtained with real world experiments: Karygiannis et al. [KAA06] measured the number of control packets using AODV and DYMO, respectively. Conducting practical experiments is also interesting as we can compare our results to results obtained previously with related protocols. DYMO is a successor to the *Ad Hoc On-Demand Distance Vector* (AODV) [PBRD03] routing protocol. AODV is also an on-demand MANET routing protocol and it works similarly to DYMO. Several experimental evaluations involving implementations of AODV have been conducted. As DYMO is a

successor to the AODV protocol and a motivating factor is to investigate how an implementation of DYMO fare when compared to results obtained using AODV. It is interesting to examine if the results obtained from an experimental evaluation of DYMO would be substantially different from the experimental results obtained with AODV.

Another motivating factor for conducting experimental evaluations is to check if a routing protocol implementation works. For example, is the implementation working according to the specification, i.e., is it able to discover routes according to the specification and to keep routes updated while they are being used? It is impossible to formally evaluate if a routing protocol implementation is working according to the specification, but it is possible to validate the correctness when setting up computers to perform the practical experiments. This is the case as one can constantly monitor the routes that have been created and if the time stamp of a route is properly updated. Furthermore, if the implementation writes to a log file, the files from each of the nodes can be examined if there is doubt with regards to correctness. Aside from correctness, the stability of the implementation is also tested when performing practical experiments. The implementation is tested in more extreme situations compared to the more gentle tests conducted during the development process when then implementation is usually not tested for extended periods.

1.4 Aim of Thesis

The focus of this thesis is on the implementation of the DYMO routing protocol and experimental evaluation of routing in MANETs in which our implementation of the DYMO protocol is deployed. The goals of this thesis are:

- *To review the DYMO Internet-Draft by implementing the specification. The implementation will be based on previously documented implementation experience.*
- *To use the implementation to evaluate the DYMO protocol with regards to various quantitative performance metrics.*

The DYMO routing protocol is defined in an IETF Internet-Draft. The draft was initially published in February 2005, and currently it has seen six revisions, the fourth being a major one which changed the packet layout. At the time of writing, the latest specification is from October 2006. Because of the frequent updates, none of the other available implementations have been updated to conform to revision later than the third. As a consequence, we do not test interoperability between different DYMO implementations in this thesis.

1.5 Methods

We will implement the fourth version of the DYMO Internet-Draft. We will make a review of some the currently available MANET routing protocol implementations and the associated challenges and methods as described in the MANET literature. As the specification is constantly being fine-tuned and updated, the DYMO specification will be implemented so that it is easy to update it to conform to newer versions of the Internet-Draft. Furthermore, the implementation will be developed such that it easy to conduct the experimental evaluations when the implementation process has finished. This means that it should be easy to query the implementation and get readings of the internal state of the implementation, for example, the content of the routing table.

We will perform practical evaluations using an experimental setup with wireless equipped laptop computers. We will survey the previous experimental evaluations that have been conducted and documented in the literature and use these as a guide when choosing topologies as well as the specific experiments to conduct.

To assist with performing repeatable practical experiments with more than a few real devices according to a predetermined topology, one can make use of evaluation testbeds for MANET protocol evaluation. A MANET evaluation testbed makes it possible to create and explore different topology scenarios even though the mobile nodes are statically placed within transmission range of each other. This makes it much easier to simulate nodes moving around, link breaks between nodes, and nodes entering and leaving the MANET. Depending on the actual testbed in consideration, the results obtained can be subject to the same reservation as mentioned when describing network simulators. How well the results compare to real world results will depend on the radio propagation model used by the evaluation testbed.

We will survey the proposed MANET evaluation testbeds. Based on the survey, one of the testbeds will be used as a part of our practical experiments. We will compare the results obtained with the testbed with the results obtained from the experiments we perform in real setup. With a real setup, we mean a setup where the nodes are physically placed so that only the nodes, which are supposed to hear each other according to the chosen topology is actually able to.

1.6 Thesis Outline

The thesis is organized into the following chapters:

Chapter 2 Mobile Ad Hoc Networks and Routing Protocols In this introduction we have not elaborated on the envisioned applications of MANETS nor have we described any MANET routing protocol in details. In chapter 2 we describe MANETs and MANET routing protocols in details and specifically give examples of the operations of three different MANET routing protocols.

Chapter 3 The DYMO Routing Protocol The DYMO routing protocol is the focus of this thesis. To be in a position to understand the following chapters we give a thorough description of DYMO, its origins, its operations, and its packet layout.

Chapter 4 Implementation Approach Several MANET protocols proposed prior to the DYMO protocol have been implemented and the challenges involved described and the implementation process has been documented. We describe the implementation challenges and give an evaluation of existing implementations with regards to the literature.

Chapter 5 DYMO-AU Design and Implementation Overview In continuation of the description and identification of implementation challenges of MANET routing protocols described in chapter 4, we present the design and implementation of our version of the DYMO routing protocol. The presentation is divided into two chapters. This chapter gives an overview of the implementation.

Chapter 6 DYMO-AU Design and Implementation Details We continue the presentation of our implementation and go into details about the design and implementation of the individual parts of the implementation.

Chapter 7 Experimental Evaluation We give an outline of experimental evaluations conducted involving the related AODV routing protocol that we have used as guidelines for our own experiments. We then describe our practical experiments and the obtained results.

Chapter 8 Conclusions and Future Work We summarize the results described in this thesis. We then conclude this thesis and list future work and research.

Central concepts in this thesis will be explained, but we assume that the reader has basic knowledge of distributed systems, wireless networks (including a basic understanding of the IEEE 802.11 MAC layer), and network protocols. Knowledge of programming languages, especially the C programming language is also assumed.

We assume that the reader has an understanding of the architecture of Unix-like operating systems and a solid understanding of system programming, including network programming, on these types of operating systems.

In chapter 5 and 6, we mention several *POSIX* system calls and library functions, deliberately without giving further references. We assume that the reader is able to use the help facilities on a Unix-like system, specifically the manual pages, to seek additional help if needed.

2

Mobile Ad Hoc Networks and Routing Protocols

In a *mobile ad hoc network* (MANET), mobile nodes communicate using wireless links without a fixed infrastructure such as base stations (access points) or centralized control. A typical mobile ad hoc network is a group of laptops operating in wireless ad hoc mode. Each mobile node acts as a router to enable multi-hop communication. A node is free to move around randomly and as a result, the topology formed by the nodes is highly dynamic and unpredictable. A MANET can operate in a stand-alone fashion, or can be connected to a fixed internetwork, for example, the Internet.

The need for multi-hop routing arises when some nodes are out of transmission range of others. A node only handles traffic within a local cloud of wireless devices. As an example consider figure 2.1, in which the circle around the nodes indicates the transmission range. For node **A** to be able to communicate with node **C** and **D** that both are out of the transmission range of **A**, the intermediate node **B** must forward packets to the other nodes.

Before moving on with the introduction to MANETs, we describe the *hidden terminal problem*; another problem that stems from nodes being out transmission range of each other. In figure 2.1, if node **A** and node **C** transmit to node **B** simultaneously, the packets may collide at node **B** causing wasted network bandwidth. The nodes **A** and **C** are hidden from each other, as they cannot sense the transmission of the other node. The use of a virtual carrier sensing mechanism can alleviate the problem. A node that wishes to transmit data sends a *request-to-send* (RTS) message before sending any data packets. The receiver then answers with a *clear-to-send* (CTS) message and other nodes hearing the RTS/CTS messages update their *Network Allocation Vector* (NAV) according to the time period specified in the messages [Gas02]. The busy state of the medium is then indicated by carrier sensing at the physical layer combined with the virtual carrier sensing information found in the NAV. In the following, we refer to the use of RTS and CTS messages as the RTS/CTS clearing procedure.

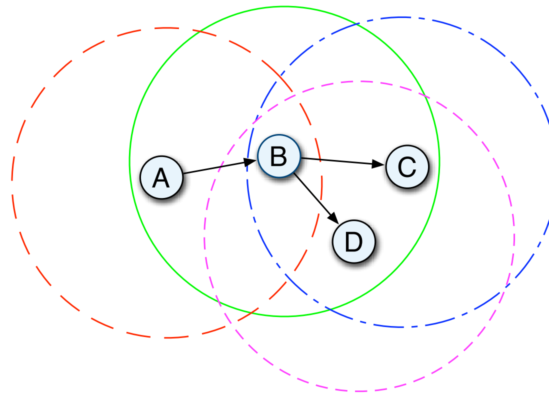


Figure 2.1: Illustration of multi-hop routing. Every node must act as a router. The circle around each node shows the transmission range. If **A** needs to reach **C** and **D**, node **B** must forward packets from **A**.

2.1 Application of MANETs

The origins of MANETs are to be found in the US military and one envisioned use is military. For example, in the battlefield, different units could be able to communicate even if an existing infrastructure has been destroyed or is untrusted. Second, MANETs could be used in rescue and disaster relief efforts, for instance in remote areas with little or insufficient communication possibilities.

A third application area is in sensor networks. A network of autonomously cooperating sensors can perform tasks not previously possible in traditional networks. Typically, nodes are relatively small units placed in an environment to monitor some kind of phenomenon. An example could be vehicle-to-vehicle communication. A sensor placed on a vehicle could detect road conditions and propagate this information to other vehicles on the road.

A fourth area is temporary networks, for example deployed at conferences, meeting rooms, and airports. Wireless Internet connection at airports can be expensive, and a group of people could share a connection with the use of a MANET.

Finally a fifth application area could be a wireless personal area network with watches, laptops, PDAs, cell phones, and wearable computing devices sharing and exchanging information and delivering added convenience for the owner.

Some of the motivation of the different application areas can be summarized as either total lack of an infrastructure, unwillingness to use any existing infrastructure, or the desire to extend coverage of an existing infrastructure.

2.2 Conventional Wired Network Routing Protocols

Most MANET routing protocols are based on or borrow ideas from conventional wired network protocols. Before we go into the details of how the different kinds

of MANET routing protocols operate and describe some specific MANET routing protocols, we make a brief overview of the types of next hop unicast routing protocols used in conventional wired networks.

2.2.1 Distance Vector Routing

In a distance vector protocol, each node maintains complete information about distances to each destination via neighbours, i.e., the next hop for that particular destination. An entry in the routing table consists of a destination address, number of hops, and next hop for the destination. When a node boots, it discovers each directly connected neighbour and initializes its routing table to contain information about these. Periodically a node sends a copy of its routing table to the directly connected neighbours.

A node **T** receiving a routing table from **S** then examines the listed entries and updates its own table according to the update received from **S**. An entry is updated if:

- **S** knows a shorter route to a node
- The hop count for a destination that **T** routes through **S** changes
- **S** advertises a destination not in the routing table of **T**

For example, figure 2.2 shows an existing routing table for node **T** (2.2a) and an update message from node **S** (2.2b) that is used to update the routing table of **T** (2.2c).

Desti-nation	Dis-tance	Next hop
A	4	R
B	7	S
C	3	S

(a) The existing routing table for **T**

Desti-nation	Dis-tance
A	2
B	5
C	3
D	1

(b) Routing table update from **S**

Desti-nation	Dis-tance	Next hop
A	3	S
B	6	S
C	4	S
D	2	S

(c) The updated routing table for **T**

Figure 2.2: The routing table message from **S** is used to update entries (destination A,B, and C) or add a new entry (destination D) to the routing table of **T**.

All distances in advertised routing tables are as seen by the sender. To compare the distance to a destination **X** going through **S**, denoted $D_{(S)}(T, X)$, to its own recorded distance, which we denote $D(T, X)$, it uses $D_{(S)}(T, X) = 1 + D(S, X)$. For example, using the tables in figure 2.2, the entry of **T** for **B** is updated to 6.

One disadvantage of distance vector routing is the possibility of routing loops. An example leading to a routing loop is depicted in figure 2.3. Initially, a link

between node **C** and **D** exists, **C** has advertised it, and consequently **B** and **A** have installed the route at distance 2 and 3, respectively. The connection between **C** and **D** then breaks, **C** updates its routing table setting the distance to **D** to infinity, but before its periodically routing table message is broadcasted, it receives a message from **B** reporting the link at distance 2. When **C** next broadcasts its routing table, advertising node **D** at distance 3, **B** updates its **D** entry and sets the distance to 4. The series of updates is shown in figure 2.4. The first row shows the entries before the link break, and the second shows them after **C** relearns the distance to **D** from **B**.

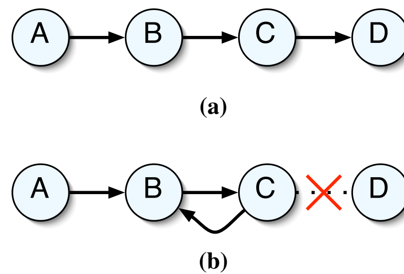


Figure 2.3: Scenario resulting in a routing loop because of the vanished connection to **D**.

A	B	C
(3,B)	(2,C)	(1,D)
(3,B)	(2,C)	(3,B)
(3,B)	(4,C)	(3,B)
(5,B)	(4,C)	(5,B)

Figure 2.4: Routing table entries (distance, next hop) with respect to **D**

The problem of the ever-increasing hop count happening after a link break is called the *counting to infinity* problem.

2.2.2 Link State Routing

Using link state routing protocols, each node maintains complete topology information about the network. The topology information is a map, represented as a graph $G = (V, E)$, i.e., the routers or nodes in the network are the nodes (V) of the graph and edges (E) are the links that connects the nodes. There is a link between two nodes if they can communicate directly.

Instead of sharing its routing table with neighbours as in a distance vector protocol, nodes share information about its outgoing links. Information about these links is obtained by periodically testing the connection to neighbours. A node sends out *Hello* messages asking whether neighbours are alive and reachable. If a

neighbour replies, the link is marked as being up, otherwise, it is marked as being down.

To let other nodes in the network know about the status (state) of its links, a node periodically floods a message with the state of each link listed. Combining all the link state advertisements received, a node builds a map or graph of the network. Using *Dijkstra's shortest path algorithm*, the shortest path to all destinations from the node can be computed. Whenever the link status changes, the routes are recomputed.

2.3 MANET Routing protocols

The routing of traffic between nodes is performed by a MANET routing protocol. MANET routing protocols can be divided into two categories. In *table-driven/proactive* routing protocols, nodes periodically exchange routing information and attempt to keep up-to-date routing information. In *on-demand/reactive* routing protocols, nodes only try to find a route to a destination when it is actually needed for communication. In the following sections, we first describe the two categories of MANET routing protocols in more details. We then list the challenges faced by MANET routing protocols.

2.3.1 On-Demand Routing Protocols

On-demand routing protocols only maintain routes that are actually used. On-demand protocols use two different operations to find and maintain routes: the route discovery process operation and the route maintenance operation. When a node wishes to communicate with some other node it tries to find a route to that node, i.e., routing information is acquired on-demand. This is the route discovery operation. Route maintenance is the process of responding to changes in topology that happens after a route has initially been created. The nodes in the network try to detect link breaks on the established routes. Examples of on-demand protocols are DSR (see section 2.5), AODV (see section 2.4), and DYMO (see chapter 3). A disadvantage of the reactive approach is that when the sending node has to discover a route to the destination, the initial delay before data is exchanged between two nodes can be long.

2.3.2 Table-Driven Routing Protocols

Proactive routing protocols maintain routing information continuously. Typically, a node has a table containing information on how to reach every other node (or some subset hereof) and the algorithm tries to keep this table up-to-date. Changes in network topology are propagated throughout the network. Examples of proactive protocols are the Topology Dissemination Based on Reverse-Path Forwarding (TBRPF) [OTL04], Highly Dynamic Destination-Sequenced Distance-Vector Routing [PB94], and OLSR (see section 2.6).

2.3.3 MANET Routing Protocol Challenges

MANET routing protocols face some challenges to be able to work in the envisioned application areas as described in section 2.1.

Dynamic topology Nodes can appear and disappear at random. Nodes can move continuously or be powered off when entering sleep mode. This means that, for example, the routing protocols cannot assume that once information is gained about the topology, it remains fixed and must consider the cost of constantly updating routing information.

Resource constraints Nodes can have limited resources with respect to computational power, e.g., RAM and CPU power available, power supply, and cost of communication. In addition, there are constraints with regard to bandwidth on the wireless link, the effects of multiple access, fading, noise, and interference conditions may severely limit the transmission rate or even prevent link establishment.

Heterogeneity Nodes can have varying characteristics with respect to the resource constraints specified above and be more or less willing to participate in routing.

To give an overview of how MANET routing protocols work, in the following, we give descriptions of how three of the many proposed protocols work. As examples of on-demand MANET routing protocols, we have chosen to describe AODV [PBRD03] (section 2.4) and DSR [JMH04] (section 2.5). AODV is the most well-known MANET routing protocol and several implementations exist [Adh]. AODV is thus far the only on-demand routing protocol promoted from an Internet-Draft to an experimental RFC. DSR is also an on-demand routing protocol. It is currently defined by an Internet-Draft, but is on its way to becoming an RFC [MANa]. DSR is interesting as it in contrast to other MANET routing protocols uses source routing. As we clarify in chapter 3, both AODV and DSR are interesting in the context of DYMO as the DYMO routing protocol can be seen as a simplified version of AODV borrowing ideas from DSR.

We then describe OLSR [CJ03] as an example of a proactive protocol. OLSR has been defined by an experimental RFC and several implementation are available [Adh]. Currently a second version of OLSR is being developed [CJ06].

2.4 The AODV Routing Protocol

The *Ad Hoc On-Demand Distance Vector* [PBRD03] (AODV) routing protocol is a reactive protocol. As is the case with all reactive ad hoc routing protocols, AODV consists of two protocol operations: Route discovery and route maintenance.

2.4.1 Route Discovery

Route discovery is the process of creating a route to a destination when a node lacks a route to it. When a node **S** wishes to communicate with a node **T** it initiates a *Route Request* (RREQ) message including the last known sequence number for **T** and a unique RREQ id that each node maintains and increments upon the sending of an RREQ. The message is flooded throughout the network in a controlled manner, i.e., a node only forwards an RREQ if it has not done so before; the RREQ id is used to detect duplicates. Each node forwarding the RREQ creates a *reverse route* for itself back to **S** using the address of the previous hop as the next hop entry for the node originating the RREQ.

When the RREQ reaches a node with a route to **T** (possibly **T** itself) a *Route Reply* (RREP), containing the number of hops to **T** and the sequence number for that route, is sent back along the reverse path. An intermediate node must only reply if it has a fresh route, i.e., the sequence number for **T** is greater than or equal to the destination sequence number of the RREQ. Since replies are sent on the reverse path, AODV do not support asymmetric links. Each node receiving this RREP creates a *forward route* to **T** in its routing table, and adds the node that transmitted the RREP in precursor list for this entry. The precursor list is a list of nodes that might use this node as next hop towards a destination. Route discovery is illustrated in figure 2.4.1, where node **2** wants to communicate with node **9** and floods an RREQ message in the network. Node **9** replies with an RREP. Intermediate nodes learn routes to both source and destination nodes via the RREQ and RREP packets.

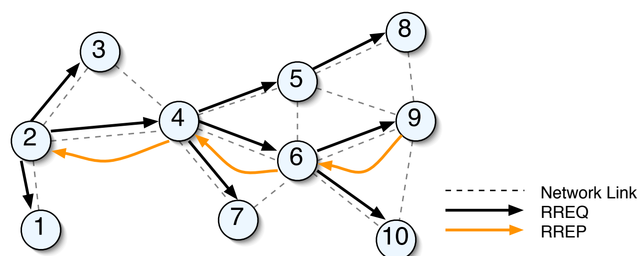


Figure 2.5: Route discovery in AODV. Node **2** wants to communicate with node **9**. Each node forwarding the RREQ creates a reverse route to node **2** used when sending back the RREP.

If an intermediate node has a route to a requested destination and sends back an RREP, it must discard the RREQ. Furthermore, it may send a gratuitous RREP to the destination node containing address and sequence number for the node originating the RREQ. Gratuitous RREPs are sent to alleviate any route discovery initiated by the destination node. It might not have received any RREQs and has not learned a route to the originator of the RREQ.

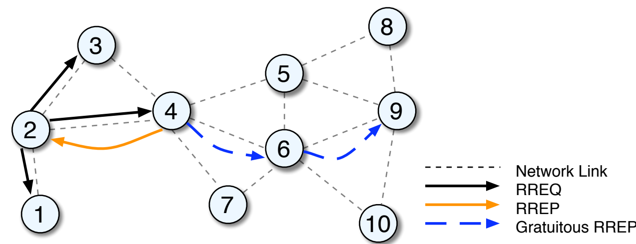


Figure 2.6: Generation of an RREP by an intermediate node. Node **4** has a route to node **9** and sends an RREP to node **2** and a gratuitous RREP to node **9**.

2.4.2 Route Maintenance

Route maintenance is the process of responding to changes in topology. To maintain paths, nodes continuously try to detect link failures (when a neighbour go out of range, the node itself moves, or some other event limiting the communication on the link). Nodes listen to RREQ and RREP messages to do this. Furthermore, each node promises to send a message every n seconds. If no RREQ or RREP is sent during that period, a *Hello* message is sent to indicate that the node is still present. Alternately, a link layer mechanism can be used to detect link failures. Beside the observation of a link failure, a node must also respond when it receives a data packet it does not have a route for.

When a node detects a link break or it receives a data packet it does not have a route for, it creates and sends a *Route Error* (RERR) packet to inform other nodes about the error. The RERR contains a list of the unreachable destinations.

If a link break occurs, the node adds the unreachable neighbour to the list. If a node receives a packet it does not have a route for, the node adds the unreachable destination to the list. In both cases, all entries in the routing table that make use of the route through the unreachable destination, are added to the list.

The list is pruned, as destinations with empty precursor lists, i.e., destinations that no neighbours currently make use of, are removed. The RERR message is either unicasted (in case of a single recipient) or broadcasted to all neighbours having a route to the destinations in the generated list. This specific set of neighbours is obtained from the precursor lists of the routing table entries for the included destinations in the RERR list.

When a node receives an RERR, it compares the destinations found in the RERR with the local routing table and any entries that have the transmitter of the RERR as the next hop, remains in the list of unreachable nodes. The RERR is then either broadcasted or unicasted as described above. The intention is to inform all nodes using a link when a failure occurs. For example, in figure 2.7, a link between node **6** and node **9** has broken and node **6** receives a data packet for node **9**. Node **6** generates a RERR message, which is propagated backwards toward node **2**.

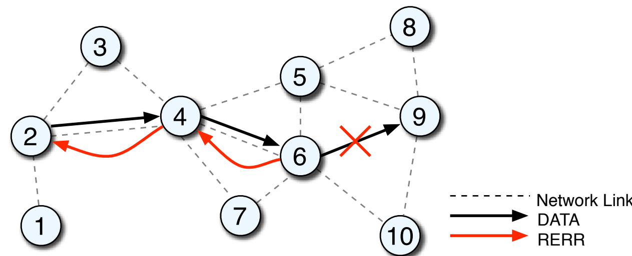


Figure 2.7: Generation of RERR messages. The link between node **6** and node **9** has broken, and node **6** generates an RERR.

To find a new route, the source node can initiate a route discovery for the unreachable destination, or the node upstream of the break may locally try to repair the route, in either cases by sending an RREQ with the sequence number for the destination increased by one.

2.5 The DSR Protocol

The Dynamic Source Routing protocol (DSR) [JMH04] allows any host to dynamically discover a source route to any destination in the network. A source route is a route that is determined by the source and correspondingly, source routing is a routing technique in which a packet is moved through a network using a path predetermined by the source node. The path information to use during the routing is placed in the packet.

As other routing protocols operating on-demand, no overhead is imposed when nodes are stationary and routes have already been created. Other characteristics that separate DSR from other on-demand routing protocols, are that source routing permits intermediate nodes to avoid keeping routing information and that it guarantees loop-free operation. It is beacon-less, i.e., it does not require exchange of periodic Hello messages. Furthermore, DSR supports unidirectional links and asymmetric routes. As all on-demand routing protocols, DSR is composed of two main mechanism: route discovery and route maintenance.

2.5.1 Basic Route Discovery

In the following, we describe the basic route discovery mechanism. The mechanism is illustrated in figure 2.8. Node **2** has a data packet to send to node **9** and floods a *Route Request* (RREQ) in the network, as it does not have a route to node **9** in its route cache. The RREQ packet contains a unique request id generated by the source node and a record listing the addresses of all intermediate nodes. The route record is initialized to the empty list by the initiator of the RREQ.

Each node receiving the RREQ rebroadcasts the packet, if the node is not the

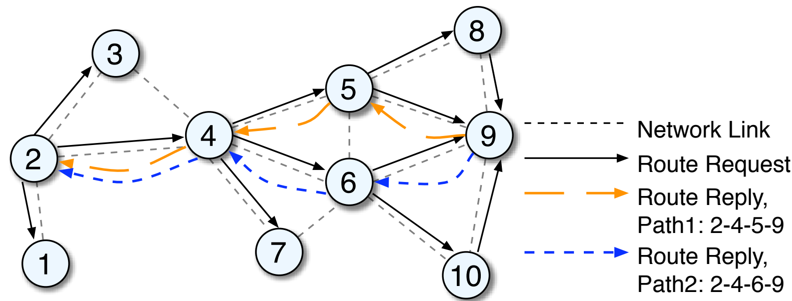


Figure 2.8: The route discovery process for DSR. Node **2** is the initiator and node **9** is the target. To make the figure more comprehensible, not all possible routes are shown. Routes involving node **8** or node **10** could for example be possible.

target, it has not forwarded the packet previously, and it does not find its own address already listed in the route record. The request id of the RREQ is used to check for already forwarded packets, i.e., duplicate RREQs. Finally, the node appends its address to the route record of the packet. In the example in figure 2.8, forwarded RREQs that would be discarded by the receiver, are not shown.

The RREQ arrives at node **9** via different routes and the node then returns a *Route Reply* (RREP) to node **2**, the initiator of the route discovery, containing the recorded route. When node **2** receives the RREP sent by node **9**, it saves the listed route in its route cache for use for subsequent sendings. The RREP can be returned various ways. If the destination node has a route to the initiator in its route cache it can use this. It can itself perform a route discovery for the initiator with the RREP packet piggybacked to avoid an infinite loop of route discoveries. Finally, the target can use the reversed sequence of hops found in the route record of the RREQ, which is illustrated in figure 2.8. In the example in figure 2.8, not all possible returned RREPs are showed. For instance, a RREQ could arrive at node **9** via node **8** and had thus taking the path **2**-**4**-**5**-**8**-**9**. The RREP could then be returned on the reverse route.

2.5.2 Route Maintenance

Each node transmitting a packet is responsible for ensuring that the next hop neighbour receives the packet. This can be performed in three ways. It can either be done listening for link-layer per-hop acknowledgements. It can be done using what is called passive acknowledgements, which means a node hears the next hop node send packets to its next hop. For example, a node **A** forwarding a packet confirms reception of the packet at the next hop neighbour **B**, when hearing **B** send the packet to node **C**, the next hop of node **B**. Finally, a flag can be set in a DSR control packet requesting explicit next hop acknowledgement.

Upon detection of a link break when forwarding a packet, a *Route Error* (RRER) error packet is sent to the node originating the packet, stating the link that is currently broken. For example, in figure 2.9, node **9** has moved outside the transmission range of node **6** and it is unable to deliver the data packet to node **9**.

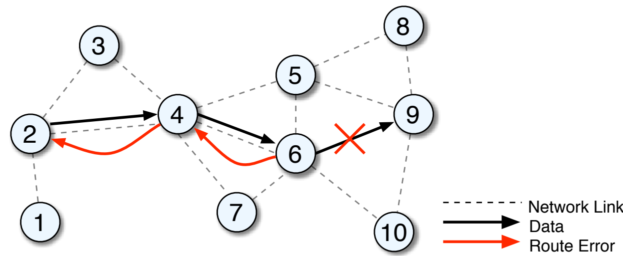


Figure 2.9: Route maintenance. Node **9** cannot be reached by node **6** anymore and a RERR is returned to node **2**.

Node **6** then returns RERR to node **4** that in return propagates it to node **2**, the original sender, which removes the route from its route cache. It can then use another cached route (for example, the path **2**-**4**-**5**-**9** learned from the previous route discovery), or perform a new route discovery for node **9**.

2.5.3 Route Discovery Optimizations

A node may extract routing information found in RREQ, RREP, RERR, and ordinary data packets that it is forwarding or overhearing while in promiscuous mode and add it to its route cache. Promiscuous mode is a state of a network adapter in which it listens to all traffic on a network, regardless of link-layer addresses. Routes learned this way, may however be subject to uni-directional links. The node can always add destinations that are on the forward direction links compared to itself. For example, in figure 2.8, if node **4** was forwarding a data packet from node **1** to node **9** it could add the presence of links from itself to node **6**, and from node **6** to node **9**. Only if the node knew that the links were bidirectional could it add the presence of reverse links, for example, in figure 2.8, from itself to node **2**. Bidirectional links could, for example, be ensured by the MAC protocol.

When an intermediate node receives a RREQ it may search its route cache for a route the target of the RREQ and return a RREP if found. The route record in the generated RREP is the concatenation of the route found in the route record of the RREQ and the intermediate nodes idea of the route to the target.

2.5.4 Route Maintenance Optimizations

A route may be shortened in a situation where a node overhears a packet it is not the intended next hop for, but its address appears in the unused part of the source route, i.e., the part that has not yet been travelled.

For example, in figure 2.10, node **6** overhears node **2** transmit a packet to node **4**. As node **6** is listed in the unused part of the source route, in this example as the next hop after node **4**, it can infer that node **4** is no longer needed when transmitting packets from node **2** to **9**.

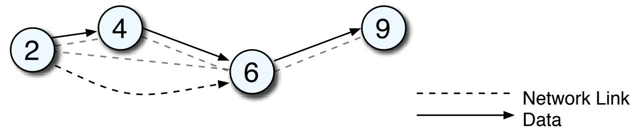


Figure 2.10: Node **6** detects that the route to node **9** can be shortened as it overhears the transmission from node **2** to **4** and it is itself listed in the source route of the packet.

Node **6** then returns a gratuitous RREP to node **2** indicating the shorter route **2-6-9**, i.e., the returned route is the original source route with the node sitting between the node sending the overheard packet and the node sending the gratuitous RREP removed.

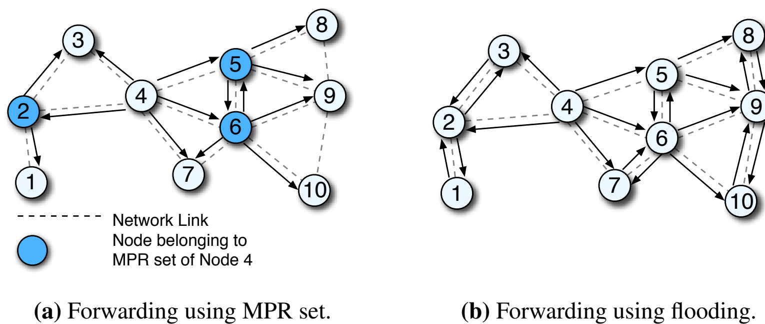
2.6 The OLSR Protocol

The *optimized link state routing* (OLSR) [CJ03] protocol is a proactive routing protocol that uses an efficient link state packet forwarding mechanism called *multipoint relaying*. The purpose of using multipoint relays to flood messages is to reduce the number of duplicate retransmission while forwarding broadcast packets. A *multipoint relay* (MPR) is a node that forward broadcast packets during the flooding process. Each node selects a subset of its 1-hop neighbours as its MPRs. We assume the links between nodes are bidirectional (symmetric). This set is called the MPR set of the node and is the set of nodes that may retransmit its messages. This set is chosen so that for every node *m* in the 2-hop neighbourhood of the node, *m* can be reached through one of the chosen MPRs. A more detailed description of how neighbours and two-hop neighbours are detected is described in section 2.6.1. The OLSR specification suggests the following MPR selection algorithm for a node **X**:

- Select from the 1-hop neighbour set ($N_1(X)$) the nodes that cover isolated nodes of the 2-hop neighbour set ($N_2(X)$), i.e., nodes in $N_2(X)$ that are only reachable from *one* node in $N_1(X)$. For example, in figure 2.11a, when selecting the MPR set of node **4**, node **1** can only be reached via node **2** and node **2** is thus added to the MPR set of node **4**.
- Until all of $N_2(X)$ is covered, select from the remaining nodes of $N_1(X)$, the node covering the highest number of nodes in $N_2(X)$. Continuing the MPR selection procedure for node **4** in figure 2.11a, node **5** and node **6** are chosen next.

The resulting MPR set of node **4** are the nodes **2**, **5**, and **6** and is depicted in figure 2.11a; all nodes in the 2-hop neighbourhood can be reached through one of the selected MPRs. Besides the MPR set, each node maintains a subset of neighbours that have selected the node as MPR. This set is called the MPR selector set.

The selection of MPR for flooding messages in the network reduces the amount of traffic in the network because of reduced broadcast message retransmissions. This is illustrated in figure 2.11. In the network in figure 2.11a, the nodes of the MPR set of node **4** (nodes **2**, **5**, and **6**) are the only ones retransmitting packets originating at node **4**. A total of four transmissions are taking place. In the network in figure 2.11b, every node forwards the packet from node **4**. A total of ten transmissions are taking place.



(a) Forwarding using MPR set. **(b)** Forwarding using flooding.

Figure 2.11: In 2.11a, node **4** has selected the shaded nodes as its MPR set. Only these nodes forward packets originating from node **4**. In 2.11b, every node retransmit the packets.

Compared to a pure link state routing protocol, two further optimizations are achieved by taking advantages of the selection of the MPR set. As explained in section 2.2.2, in a traditional link state routing protocol each node broadcasts messages that list the state of each its links. Because of how the MPRs have been selected only nodes chosen as MPRs need to publish their link state. Thus, the first optimization is achieved as the number of control packets flooded in the network is reduced.

An MPR node may choose to only report partial link state information, namely the MPR selector set of the node. As a consequence, the size of control packets is reduced. This gives the second optimization. Given the link state messages containing the information about the MPR selector set of all the MPRs, the nodes in the MANET have enough information to calculate shortest path routes to all hosts.

Messages containing link state information are called *Topology Control* (TC) packets. TC packets are periodically flooded in the network and as described previously only the MPR subset broadcast the TC packets and only the MPR selector set of an MPR node is announced in the TC packet.

2.6.1 Neighbour Discovery

To obtain the necessary topology information for a node to be able to select its MPR set, it is necessary to get information about its one- and two-hop neighbours. In addition, information about nodes that have selected a specific node as MPR, the MPR selector set, must be maintained. This information is obtained by periodically sending HELLO messages in which nodes exchange information about their neighbours, the link state, and type of neighbours.

To initially discover neighbours, a node first sends an empty HELLO message. The process is illustrated in figure 2.12, where two nodes, **A** and **B**, exchange HELLO packets. In the figure, node **A** first sends an empty HELLO packet. Node **B** receives the packet and in the next sent HELLO message, it includes the address of **A** and marks it as an asymmetric neighbour, which means that **B** has not found its own address in the HELLO packet received from **A**. When **A** receives the packet from **B** and finds its own address in the HELLO packet coming from **B**, it sends a HELLO packet announcing node **B** as a symmetric neighbour. Finally, node **B** sends a HELLO packet where **A** is listed as a symmetric neighbour.

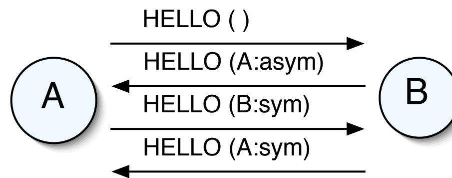


Figure 2.12: Neighbour discovery session.

As a HELLO message contains a list of neighbours of the sending node, a node receiving the message can maintain its two-hop neighbour set based on this list. All symmetric neighbours advertised in the HELLO packet, not including the node itself, is added to the receiver's two-hop neighbour set. This implies the inclusion of one-hop neighbours, however, these are excluded when calculating the MPR set. For each advertised neighbour in a HELLO message, a flag indicates if the sending node has chosen this neighbour as an MPR. Upon receiving a HELLO packet a node checks the list of neighbours and compares with its own address. If a match is found, the sender of the HELLO message is added to the MPR selector set of the receiver.

3 The DYMO Routing Protocol

The *Dynamic MANET On-demand* DYMO routing protocol is a newly proposed protocol currently defined in an IETF Internet-Draft [CP06b] in its sixth revision and is still work in progress. The version of the protocol described here is the fourth version as our implementation of DYMO presented in chapter 5 and chapter 6 is based on this version.

DYMO is a successor of the AODV routing protocol [PBRD03] and is the current engineering focus for reactive routing in the IETF MANET working group [PBRDC]. It operates similarly to AODV, which we described in section 2.4. DYMO does not add extra features or extend the AODV protocol, but rather simplifies it, while retaining the basic mode of operation.

DYMO is not the first attempt to make an enhanced version of AODV. *AODV with Path Accumulation* (AODV-PA) proposed by Gwalani et al. [GBRP03] extends AODV with the source route path accumulation feature of DSR. As described in section 2.5.1, in the DSR protocol the addresses of intermediate nodes are accumulated in the DSR RREQ and RREP packets when they are disseminated in the network, i.e., every node forwarding an RREQ or RREP adds its own address to the packet. In this way, nodes also learn about routes to other nodes in the network. AODV-PA makes no other modifications to AODV. Gwalani et al. found that under conditions of high-load and moderate to high mobility, AODV-PA has improved performance compared to AODV and also found it scaled better in large networks. The results were obtained using the ns-2 simulator.

AODVjr proposed by Chakeres and Klein-Berndt [CKB02] removes all but the essential elements of AODV. This means sequence numbers, gratuitous RREPs, hop count, Hello messages, RERR messages, and precursor lists are omitted compared with AODV. Furthermore, route lifetimes are only updated when receiving packets. In AODV, route timeouts are also updated when sending packets. What remains, is basic route discovery using RREQ and RREP where only the destination responds to RREQs. To maintain active routes the destination is required to periodically send *connect* messages if traffic is unidirectional. Using the ns-2 simulator, Chakeres and Klein-Berndt found that AODVjr achieves nearly the same

performance as AODV, but has much lower control traffic overhead. In addition, Chakeres and Klein-Berndt estimates the combined implementation and debugging effort of AODVjr to take less than half the time compared with a full AODV implementation. They also note that AODVjr could easily be extended with, for example, link layer feedback, RERR messages, and sequence numbers.

Using AODV as a basis, DYMO combines the ideas originated in AODV-PA and AODVjr. From AODV-PA (and DSR), it borrows path accumulation. As AODVjr it removes features that may be regarded as extensions to the core functionality: as AODVjr, DYMO removes gratuitous RREP, precursor lists, and Hello messages. Hello messages are not mandated by the AODV specification either, but compared to the DYMO specification, their use and packet layout are specified. Compared to AODVjr, DYMO retains sequence numbers, hop count, and RERR messages. As of the fifth version of the DYMO Internet-Draft [CP06c], the use of hop count has been made optional.

In the rest of this chapter, we give a detailed description of the DYMO routing protocol. The description gives more low-level details compared to the descriptions of OLSR, DSR, and AODV given in chapter 2. These details are given to provide a better prerequisite to understand our implementation of DYMO, which we describe in chapter 5 and 6. Section 3.1 gives a short overview of DYMO besides the one already given. In section 3.2, the route discovery process is explained and in section 3.3, we explain the route maintenance process. Finally, in section 3.4 we describe the packet format used in DYMO.

3.1 Protocol Overview

As is the case with all reactive ad hoc routing protocols, DYMO consists of two protocol operations: route discovery and route maintenance. Routes are discovered on-demand when a node needs to send a packet to a destination currently not in its routing table. A route request message is flooded in the network using broadcast and if the packet reaches its destination, a reply message is sent back containing the discovered, accumulated path.

Each node maintains a routing table with information about nodes. Each entry in the routing table consists of the following fields:

Destination Address: the IP address of the destination

Sequence Number: the destination sequence number

Hop Count: number of hops towards the destination

Next Hop Address: the IP address of the next node on the path towards the destination

Next Hop Interface: the interface used to send packets towards the destination

Is Gateway: flag that specifies if the destination node is an Internet gateway

Prefix: number that indicates if the destination address is a network address

Valid Timeout: the time at which the route table entry is no longer valid

Delete Timeout: the time after which the route table entry must be deleted

The prefix value advertises connectivity to a subnet of nodes within its address space, i.e., it is a network address, rather than a host address. For example, if the prefix value is 24 and the address of the destination is 192.168.42.50, the node can forward packets to nodes having addresses with the prefix 192.168.42.X.

Each node must maintain its own sequence number. The sequence number is incremented each time the node sends a route request message. This allows other nodes to determine the order of discovery messages to avoid stale routing information, to detect duplicate messages, and to ensure loop freedom.

3.2 Route Discovery

Route discovery is the process of creating a route to a destination when a node needs a route to it. When a node **S** wishes to communicate with a node **T**, it initiates a *Route Request* (RREQ) message. The RREQ message and the *Route Reply* (RREP) message, which we describe later in this section are collectively known as *Routing Message*s (RM) because they are used to distribute routing information. The sequence number maintained by the node is incremented before it is added to the RREQ. We illustrate the route discovery process using figure 3.1 as an example. In the figure, node **2** wants to communicate with node **9** and thus, node **2** is **S**, the source, and node **9** is **T**, the target destination.

In the RREQ message, the node **2** includes its own address and its sequence number, which is incremented before it is added to the RREQ. It can also include a prefix value and gateway information if the node is an Internet gateway capable of forwarding packets to and from the Internet. Finally, a hop count for the originator is added with the value 1. Then information about the target destination **9** is added. The most important part is the address of the target. If the originating node knows a sequence number and hop count for the target, these values are also included. To sum up, the RREQ so far contains information about node **2** that originated the RREQ and node **9**, the target destination. The addresses in an RREQ message are grouped together in what is called an address block and the associated attributes are grouped together in a TLV block.

The message is flooded using broadcast, in a controlled manner, throughout the network, i.e., a node only forwards an RREQ if it has not done so before. The sequence number is used to detect this. Each node forwarding an RREQ may append its own address, sequence number, prefix, and gateway information to the

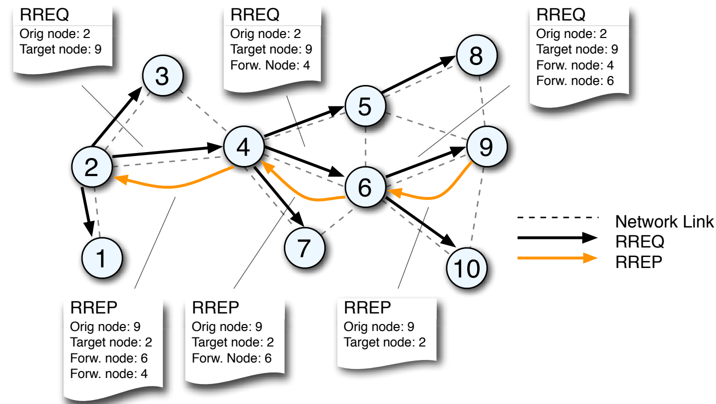


Figure 3.1: The DYMO route discovery process. Node **2** wants to communicate with node **9**. Each node forwarding the RREQ creates a reverse route to **2** used when sending back the RREP. When sending back the RREP, nodes on the reverse route create routes to node **9**.

RREQ, similar to the originator node. Upon sending the RREQ, the originating node will await the reception of an RREP message from the target. If no RREP is received within RREQ_WAIT_TIME, the node may again try to discover a route by issuing another RREQ. RREQ_WAIT_TIME is a constant defined in the DYMO specification and the default value is 1000 milliseconds. In figure 3.1, the nodes **4** and **6** append information to the RREQ when they propagate the RREQ from node **2**.

When a node receives an RREQ, it processes the addresses and associated information found in the message. The information for a node **I**, is compared with the corresponding entry in the routing table of the node, if one exists. The information about the originator found in the RREQ is processed first, but subsequent entries are processed the same way:

- If the routing table does not contain an entry for the originator, one is created. The next hop entry is the address of the node from which the RREQ was received. Likewise, the next hop interface is the interface on which the RREQ was received.
- If an entry exists, the sequence number and hop count found in the RREQ is compared to the sequence number route and hop count in the table entry to check if the information in the RREQ is stale or should be disregarded.
- If an entry exists and is not stale or disregarded, the entry is updated with the information found in the RREQ.

If the originator entry in the RREQ is found to be stale or disregarded, the RREQ is dropped. For other nodes, the information is removed from the RREQ.

If an RREQ is not dropped, each node processing the RREQ can create *reverse routes* to all the nodes for which addresses are accumulated in the RREQ.

Upon adding an entry for **I** to its route table entry, the node processing the RREQ increments the hop count value for **I** found in the RREQ to correctly reflect the number of hops the RREQ has travelled since the node **I** added its own information to the RREQ.

When the RREQ reaches the destination node **9**, it processes the packet similar to nodes that have forwarded the packet, and uses the information accumulated in the RREQ to add route table entries. Specifically, an entry for the node **2** that originated the RREQ, is installed.

An RREP message is then created as a response to the RREQ, containing information about node **9**, i.e., address, sequence number, prefix, and gateway information, and the RREP message is sent back along the reverse path using unicast.

Since replies are sent on the reverse path, DYMO does not support asymmetric links. The packet processing done by nodes forwarding the RREP is identical to the processing that nodes forwarding an RREQ perform, i.e., the information found in the RREP can be used to create *forward routes* to nodes that have added their address block to the RREP.

We shortly summarize the route discovery process depicted in figure 3.1: Node **2** wants to communicate with node **9** and floods an RREQ message in the network. As can be seen in the figure, when node **2** begins route discovery, the RREQ initially contains the address of the originator and target destination. When node **4** receives the RREQ, it installs a route to node **2**. After node **4** has forwarded the RREQ, it has added its own address to the RREQ, which means it now contains three addresses. Identical processing occurs at node **6** and it installs a route to node **2** with a hop count of 2 and node **4** as the next hop node.

When node **9** receives the RREQ, it contains four addresses and has travelled three hops. Node **9** processes the RREQ and install routes using the accumulated information and as it is the target of the RREQ, it furthermore creates an RREP as a response. The RREP is sent back along the reverse route. Similar to the RREQ dissemination, every node forwarding the RREP adds its own address to the RREP and install routes to node **9**.

3.3 Route Maintenance

Route maintenance is the process of responding to changes in topology that happens after a route has initially been created. To maintain paths, nodes continuously monitor the active links and update the Valid Timeout field of entries in its routing table when receiving and sending data packets. If a node receives a data packet for a destination it does not have a valid route for, it must respond with a *Route Error* (RERR) message.

When creating the RERR message, the node makes a list containing the address and sequence number of the unreachable node. In addition, the node adds all entries

in the routing table that is dependent on the unreachable destination as next hop entry. The purpose is to notify about additional routes that are no longer available. The node sends the list in the RERR packet. The RERR message is broadcasted.

The dissemination process is illustrated in figure 3.2. A link between node **6** and node **9** breaks and node **6** receives a data packet for node **9**. When we say a link is broken, it could just be that the time stamp in the route table entry for a node timed out and the entry has become invalid. Node **6** generates an RERR message, which is propagated backwards towards node **2**.

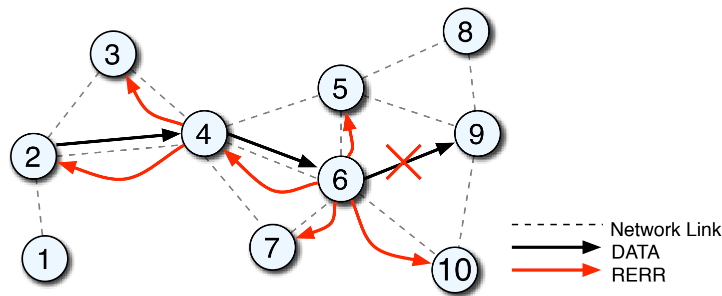


Figure 3.2: Generation and dissemination of RERR messages. The link between nodes **6** and **9** breaks, and node **6** generates an RERR. Only nodes having a route table entry for node **9** propagate the RERR message further.

When a node receives an RERR, it compares the list of nodes contained in the RERR to the corresponding entries in its routing table. If a route table entry for a node from the RERR exists, it is invalidated if the next hop node is the same as the node the RERR was received from and the sequence number of the entry is greater than or equal to the sequence number found in the RERR. If a route table entry is not invalidated, the corresponding entry in the list of unreachable nodes from the RERR must be removed. If no entries remain, the node does not propagate this RERR further. Otherwise, the RERR is broadcasted further. The sequence number check mentioned, is performed to only invalidate fresh routes and to prevent propagating old information. The intention of the RERR distribution is to inform all nodes that may be using a link, when a failure occurs. RERR propagation is guaranteed to terminate as a node only forwards an RERR message once.

In figure 3.2, when the RERR is broadcasted, additional nodes beside node **4** and **2** will receive the message, for example, the nodes **5**, **7**, and **10**. As none of these use node **6** as a next hop towards node **9**, they all drop the RERR after processing the message.

In addition to acting upon receiving a packet to a destination without a valid route table entry, nodes must continuously try to detect link failures to maintain active links. Link failures occur, for example, when a neighbour go out of range, the node itself moves, or some other event limiting the communication on the link.

The mechanisms used by a node to monitor active links can be Hello messages, link layer feedback, neighbour discovery, or route timeouts. Hello messages are packets that are periodically broadcasted with the intent of detecting the presence or disappearance of neighbours. However, the fourth revision DYMO specification draft, does not specify the use or packet layout of Hello messages. Nor is neighbour discovery explained in details. As of the fifth revision of the DYMO specification draft, the use of Hello messages and the unspecified neighbour discovery have been updated to suggest the use of neighbourhood discovery as specified in the MANET *Neighborhood Discovery Protocol* (NHDP) [CDD06b].

If a broken link is detected, the node may disseminate an RERR to notify other nodes about the broken link. The process is identical to the one described above. Finally, when a node receives an RERR for a destination, to rediscover a route, the node can initiate a route discovery for the unreachable destination by sending an RREQ message.

3.4 Generalized Packet and Message Format

As of revision 4 of the DYMO specification [CP06b], the packet format conforms to the *generalized MANET packet and message format* [CDD06a]. As the generalized message format strives to be extensible and flexible and to represent addresses in compact way, there is no fixed binary layout of messages. Various fields may or may not be present depending on flags set in the message and addresses and associated attributes may be present in the message using a number of different layouts. A set of addresses in the message is represented in an address block and the associated attributes in a *TLV* (type-length-value triplet) block that follows right after the address block.

Using a DYMO RREQ packet as an example, we describe the parts of the generalized message format applicable to DYMO in details in the following. With regards to DYMO, a message consists of a message header and zero or more occurrences of address block and TLV block pairs.[1] As a concrete example of a packet using the generalized message layout, a DYMO RREQ packet is depicted in figure 3.3. The first eight octets of the message is the message header and the following octets consist of an address block and an associated TLV block. The fields of the message are explained in further details below.

3.4.1 The Message Header

A message header consists at least of six octets and the following fields: msg-type, msg-semantics, msg-size, and msg-tlv-block-size. The example packet shown in figure 3.3 has the type DYMORREQ and a specified size of 29 octets, i.e., the size

[1] In the general case, a message consists of a message header, a message TLV block, and zero or more occurrences of address block and TLV block pairs. In DYMO, the TLV block following the message header, is not used and the mandatory tlv-block-size is regarded to be a part of the message header.

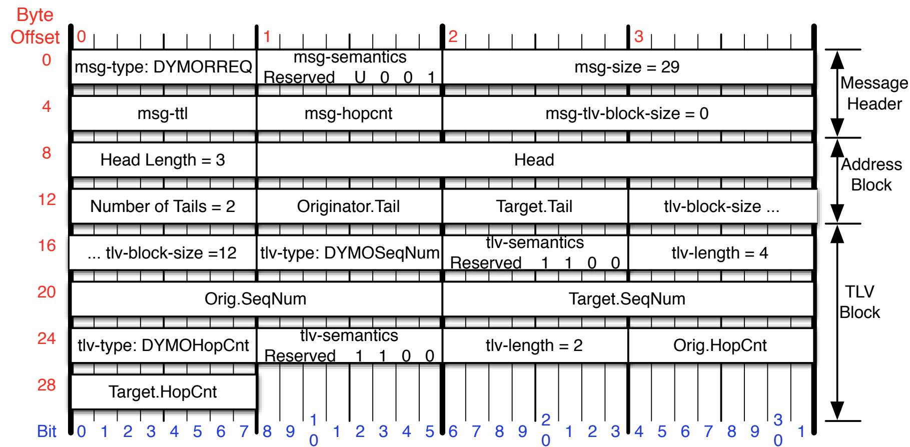


Figure 3.3: DYMO Route Request message example. The message contains information about two nodes.

of both the message header and the following message body. The msg-semantics fields specifies the interpretation of the rest of the message header as it can optionally include a field with the address of the originating node, a ttl field, a hop count field, and finally a sequence number field. The various bits in the msg-semantics bit-field are to be interpreted in the following way (the numbers refer to figure 3.3):

Bit 12 If set, the message must *not* be forwarded if the message type is unknown to the recipient.

Bit 13 Specifies semantic of the sequence number if included. If set, the sequence number is type dependent, meaning that the message source maintains sequence numbers for each possible message type. If bit 15 is set, i.e., if no sequence number is included in the message header, this bit must be cleared.

Bit 14 If set, ttl and hop count fields are *not* included.

Bit 15 If set, originator address and sequence number fields are *not* included.

In figure 3.3, we see that following the msg-size fields, the message header includes ttl and hop count fields. This is because bit 14 of the msg-semantics field is 0, which means that these two fields are included. Similar, bit 15 is set, which means that the originator address and sequence number are not included. Bit 13 is cleared as the message header contains no sequence number. No value for bit 12 (the U-bit), is specified. The msg-tlv-block-size field following the hop count field is 0, which means that no TLV block follows the message header.

3.4.2 The Message Body

The message body of a DYMO RREQ message consists of a number of addresses and attributes associated with these addresses. The addresses are contained in one or more address blocks and the attributes are contained in a TLV block following each individual address block.

The Address Block Assuming an address can be specified as a sequence of bits of the form *head:tail*, to represent a set of addresses in a compact form, the longest sequence of leftmost octets, the *head*, can be shared and have different *tails*, i.e., the distinct part of the addresses. Thus, addresses contained in an address block share the same head and have different tails.

As explained earlier in section 3.4.1, when an RREQ is initially created, it contains the address of the originator and the destination as well as sequence number and hop count. In figure 3.3, we see an example of an address block containing the addresses of the originator and the destination (target). First appearing at byte offset 8 in the figure, is a Head Length field specifying that the head is 3 octets long. The next 3 octets give the actual value of head. Then a number of tails fields that in this case specifies that two tails follow. When concatenated with the Head field, the *Originator.Tail* field represents the IP address of the originator of the RREQ, the *Target.Tail* field represents the destination target node of the RREQ.

As an example, if the three octets of the head were 192, 168, and 42 and the Originator.Tail field and Target.Tail field contained 50 and 51, respectively, the IP address of the originator of the RREQ would be 192.168.42.50 and IP address of the destination target would be 192.168.42.51.

The TLV block After the address block, an address TLV block follows. A TLV block consists of a tlv-length field, and zero or more occurrences of TLVs (type-length-value triplets). In figure 3.3, the tlv-block-size fields specifies that the length of the TLVs that follows are 12 octets. Returning to a TLV, it consists of the following two fields: tlv-type and tlv-semantics. Depending on the value of the tlv-semantics bit-field, it can furthermore include the fields: tlv-length, tlv-index-start, tlv-index-stop, and tlv-value. The length field gives the length in octets of the data present in the tlv-value field. The two index fields allow the TLV to only apply to a sequence of the addresses. The overall goal of this flexible message layout is to let a TLV apply to a varying number of addresses in the address block. The bits of the tlv-semantic bit-field have the following interpretation (numbering begins with the least significant bit in the bit-field, the rightmost bit in individual octets in figure 3.3):

Bit 0 The novalue bit; if set, the TLV contains no length and value fields.

Bit 1 The extended bit; if set, the size of the field is 16 bits (as opposed to 8 bits).

Bit 2 The noindex bit; if set, the TLV the index-stop and index-start elements are not included.

Bit 3 The multivalue bit; if set, the TLV includes separate values for each of the specified addresses.

Looking at the DYMO RREQ packet in figure 3.3, we see bit 2 and bit 3 is set for both TLVs. Consequently, the TLVs include a length and value field, the length field of the TLVs is 8-bit, the TLVs includes no index-start and index-stop fields, and finally the TLVs include separate values for all addresses.

The first TLV has the type DYMOSeqNum and a specified length of 4 octets for the value field. As the associated address block had two entries, the field is divided into two equal-sized fields. The first gives the sequence number for the first address in the address block, the originating node, and the last value field gives the sequence number for the second address in the address block, which in this case is the last known sequence number for the target.

The second TLV has the type DYMOHopCnt and a specified length of 4 octets for the value field. Again, the field is divided into two equal-sized fields. The values specify the hop count for the two addresses in the address block.

4 Implementation Approach

The list of ad hoc routing protocols at Wikipedia [Adh] has 24 entries for on-demand routing protocols. The same page lists real world implementations for only 6 of the above 24 protocols (AODV, DSR, DYMO, LQSR, LUNAR, and TORA). Few proposed on-demand ad hoc routing protocols have ever been implemented. Consequently, a lot of the research with regards to testing and performance measurements taking place within the MANET field is done using simulators.

One of the stated design goals for the DYMO routing protocol is that it should be possible to do a simple, quick implementation [BRCJP04]. Even though the DYMO protocol is simpler than, for example, AODV, the simplicity is primarily obtained in the routing logic layer while the challenges and problems outlined in this chapter (section 4.1.1), for example, to identify the need for route discovery and buffer packets during route discovery, are just as valid for DYMO as they are for AODV.

The inherent problems associated with the implementation of on-demand ad hoc routing protocols makes it cumbersome to implement these protocols because of the system-level programming required to address the problems. Features looking feasible to implement on paper can also be a lot harder to implement in reality. One example is link layer feedback and overhearing packet transmission as used by DSR.

Several frameworks to help ease the complexity of implementing on-demand routing protocols have been proposed [KNSW02, Car03, KZG03]. So have frameworks that provide a common API for communications protocols on different platforms [CM03, AGSI02]. Of these five frameworks, only the *Ad-hoc Support Library* (ASL) [KZG03] and PICA [CM03] are publicly available. Some of the problems encountered when implementing on-demand ad hoc routing protocols that is described in this chapter, is well addressed by ASL and we considered using it for our implementation of DYMO. However, ASL has not been updated for three years and it is uncertain if it works with the Linux 2.6 series of kernels. Accordingly, we chose not to use the library, but the implementation issues and solutions discussed by the ASL authors and implemented in ASL are discussed several times in this

chapter. Because of lack of availability or maintenance, none of these frameworks have found widespread use. For example, ASL has been used in one additional implementation [Ara] aside from an AODV implementation and a DSV implementation developed by the ASL authors. Hopefully, as more on-demand ad hoc routing protocol implementations are created and the implementation experience documented, the time that must be invested to implement a new routing protocol can be reduced.

The outline for the remainder of the chapter is as follows. In section 4.1, we first give an overview of the challenges implementers face when implementing an on-demand ad hoc routing protocol. These challenges emerge, as the on-demand routing model does not easily fit into the standard operating system routing and packet forwarding model. We describe the problems with the routing model of current operating systems and identify the necessary extra events that must be recognized to ensure correct behaviour of on-demand ad hoc routing protocols.

Then, in section 4.2, we describe and discuss the different design strategies that have previously been deployed in implementations of on-demand ad hoc routing protocols, focusing on AODV implementations. The intention is to give an overview of the developed solutions and point out best practices and experiences learnt.

4.1 Challenges

When we are discussing the challenges faced when implementing an on-demand ad hoc routing protocol, it is of relevance to recap the routing architecture of current operating systems. In particular, how the functionality is divided and why implementing on-demand protocols is a challenge compared to implementing traditional routing protocols or proactive ad hoc routing protocols.

Using terminology defined by Kawadia et al. [KZG03], the routing functionality is separated between the *packet forwarding* function, and *packet routing* function. The packet forwarding function consists of the route selection algorithm within the TCP/IP stack of the operating system kernel. When the IP-layer receives a packet, it inspects a table called the forwarding table. Based on the IP destination address it determines if the packet should be directed to an outgoing network interface, discarded, or delivered to a local application.

The packet routing function is the process of controlling the kernel routing table, populating it with entries to destinations using the optimal route. The definition of an optimal route is dependent on the routing algorithm; the number of hops to the destination is usually the chosen metric. The program performing the routing is typically implemented in user space as a program running in the background (the routing daemon).

In the above we distinguish between the forwarding function and the routing function, but as Kawadia et al. note, the terms are used interchangeably and they are often just called the kernel routing table. For example, in Linux the routing

table populated by the packet routing function is called the *Forwarding Information Base* (FIB) routing table. The naming in Linux of this table as the FIB table is misleading, as this table is called the Routing Information Base (RIB) table in other operating systems. The packet forwarding function uses this table and another table called the routing cache [RGK+04]. From a user's point of view, it is rarely necessary to be able to distinguish between these two tables and a user usually only needs to manipulate the FIB routing table, i.e., the routing table controlled by the packet routing function. This is the table operated on by default by the `route` and `ip` utilities used for routing table manipulation on Linux.

On Linux the route selection process is carried out the following way: when selecting a route for a packet, the kernel first searches the routing cache for an entry matching the destination IP address of the packet, and if found it forwards the packet to the next hop specified in the routing cache entry. Entries that have not been used for some time expire and will be deleted. If no entry for the destination is found in the routing cache, the kernel makes a look-up for the destination in the kernel routing (FIB) table using longest prefix matching [Bro03]. If an entry is found in the table, a new entry for the destination is created and inserted into the routing cache, i.e., the kernel routing table is used to populate the routing cache.

Working under the above described network stack architecture poses no problems for implementations of ordinary routing protocols like OSPF [Moy98] and RIP [Mal98]. The same is true for implementations for proactive ad hoc routing protocols. In these protocols, participating nodes broadcast complete or partial information about nodes that can be reached from the node. The received info is then used when maintaining the kernel routing table, adding and deleting entries when new routes to destinations are learned.

For on-demand ad hoc routing protocols the problem is as follows: if a user space application wants to communicate with a host for which there exists no route, an error is returned signalling that the destination host is unreachable. The failed connection attempt is never registered anywhere except in the program trying to establish the connection. Using on-demand ad hoc routing protocols, routes may not be known in advance, and the routing daemon must be notified about the connection attempt to be able to discover a route to the destination. Furthermore, packets must be buffered while route discovery is ongoing.

4.1.1 Identifying the On-demand Ad Hoc Routing Challenges

Chakeres and Belding-Royer [CBR04] have examined and identified the support currently unavailable in operating systems needed to implement AODV. Kawadia et al. [KZG03] have a similar evaluation for general on-demand ad hoc routing protocols and identify some of the same needed support. Besides the items also treated by Chakeres and Belding-Royer, Kawadia et al. add some additional items as their goal is to build a framework to help ease the implementation of a wider range of on-demand routing protocols, e.g., protocols mixing forwarding and routing functions (DSR) that extend on the support needed by AODV.

The list below is based on the list given by Chakeres and Belding-Royer [CBR04].

1. When to initiate a route discovery cycle: An implementation of an on-demand MANET routing protocol must intercept route requests from application programs to detect if a route to a currently unknown host is requested. The problem of the current network stack architecture is that we only know we need a route after the packet has already crossed the boundary between user space and kernel space.

2. When and how to buffer packets during route discovery: Packets destined for a host with an unknown destination should be buffered while the route discovery is in progress. If a route is found, the packets should be reinserted into the IP layer and sent to the destination. If a route is not found, the packets should be discarded and the application program should be notified.

3. When to update the lifetime of an active route: As a part of the routing protocol logic, routes that have not been used for a certain amount of time are deleted. When receiving, sending, or forwarding to a known destination, timeouts must be updated.

4. When to generate an RERR if a valid route does not exists: Under normal operation the IP layer discards packets for which no valid route table entry exists and return a ICMP destination host unreachable message. Instead, the routing daemon should be notified about the event.

In the list given by Chakeres and Belding-Royer, a fifth event that must be supported is mentioned: when should a node generate route error message after sequence number loss? A node loses its sequence number if it reboots or the routing daemon is restarted. It must generate a route error message if other nodes use the node as a router after daemon restart to avoid routing loops. In the above list, we have omitted this item. With the ability to identify packets for destinations with no valid route and some sort of support for timed operations in the routing protocol implementation, reception of data packets during this period can be handled in user space. As we wish to emphasize the support missing from operating systems in order to implement on-demand routing protocols, we find that there is no inherent need for the operating system to identify this event.

4.2 Implementation Techniques on Linux

In this section, we give an overview of the different implementation techniques that have been used to address the challenges outlined in section 4.1.1. Special emphasis is given on implementation techniques for Linux. We highlight the advantages and disadvantages of each technique and discuss how the implementation techniques and the techniques available for programmers have evolved.

The alternatives described in this section are:

Kernel Modifications Modify the source code of an operating system kernel

Snooping Use packet capturing facilities

Netfilter Use the netfilter framework of the Linux TCP/IP stack

Of the listed three opportunities, netfilter is available for Linux only. For one to make modifications to an operating system kernel requires access to the source code and a license that allows redistribution. Source code for a host of operating systems, including permissive licenses, is readily available from different sources, the most noticeable being Linux and the BSD family of operating systems.[1]

Much of the source code for the operating system Windows CE is available, but parts of it, including the TCP/IP stack is only available through a non-disclosure agreement [Wes03]. However, any modified source code cannot be redistributed without a special license from Microsoft making the approach less attractive. Further weaknesses of the kernel modification approach are examined in section 4.2.1.

The introduction of the netfilter packet filtering framework [MBR$^+$] (see also section 4.2.3) in the 2.4 series of Linux kernels and its subsequent adoption has extended the way implementations can be designed and made it easier to implement ad hoc routing protocols on Linux. For example, the AODV-UCSB implementation originally used the kernel modification techniques, but the authors reported that the implementation “suffered from some intermittent problems” [CBR04] and later switched to netfilter for a later version of the implementation.

The Linux netfilter framework can also partly be attributed to the fact that Linux has become the most popular platform for on-demand ad hoc routing protocol implementation development: Of the thirteen listed implementations on the AODV web page [PBRDC], eight are for Linux only.[2] Similar results can be found in a MANET implementation list maintained by Kuladinithi [Kul05] that lists twice as many AODV implementations for Linux as for Windows, the second most popular choice.

The implementations announced for DYMO, excluding implementations for the network simulators *ns-2* [BEF$^+$00] and *OPNET* [OPN], have so far been for Linux [RR, KBb].

To meet the challenges identified in section 4.1.1, there are additional choices influencing the functioning of an implementation, besides the previous three mentioned approaches. These are:

Packet Buffering When awaiting the result of a route discovery, are outstanding packets buffered in user space or kernel space?

Kernel or User Space Is the solution implemented entirely in user space or entirely in kernel space or using some hybrid approach?

[1] FreeBSD, OpenBSD, and NetBSD.

[2] Kernel AODV, AODV-UU, AODV-UIUC, AODV-UCSB, AODV for IPv6, HUT-AODV, MAODV-UMD, and Mad-hoc.

Interaction with Forwarding Logic Does the implementation also interacts with forwarding?

These considerations are neither independent of each other or the above-mentioned approaches. For example, packet buffering in user space is meaningless with a kernel space only implementation. The three implementation choices are treated in section 4.2.4 and additionally when pertinent in the following sections.

4.2.1 Kernel Modification

By modifying the networking code of an operating system kernel, it is possible to determine the required events identified in section 4.1.1. Royer and Perkins modified the Linux kernel to support their implementation of the AODV protocol [RP00]. The kernel source was modified so that a route look-up failure would result in a notification to a user space daemon that was a part of the implementation. Other modifications made were support for updating of route timeouts in the kernel routing table, buffering of outstanding packets, and detection of duplicate protocol control packets. All achieved by modifying the Linux kernel networking code.

The advantage of this approach is that modifying the kernel source code the events are explicitly determined. Also by modifying the kernel data structures and support code directly, there is no overhead of additional protocol accounting, compared to a user space implementation or even a Linux kernel module. Disadvantages are difficult installation and maintainability: To use the implementation requires setting up and compiling the whole Linux kernel source tree, which can be a complex task. Regarding maintainability, patches (modifications) might only apply cleanly against a certain version of the Linux kernel. There could even be problems with kernels with the same version number as distributions apply their own set of patches to the Linux kernel source. Finally, as Royer and Perkins note:

> Understanding the Linux kernel and network protocol stack requires examining a significant amount of undocumented, complex code.

The first release of the AODV-UCSB implementation used the kernel modifications approach [RP00]. Desilva and Das also made an implementation of AODV by modifying the Linux kernel [DD00] and the in-kernel ARP implementation. An implementation of the DSR routing protocol [JMH04] complying with the first public draft of the specification also used the kernel modification strategy [MBJ99]. The modifications were based on FreeBSD 2.2.7 and 3.3 kernels.

4.2.2 Snooping

Using code built into the kernel of most operating systems, a user space program can capture all incoming and outgoing packets on a network interface [SFR03] and identify the events mentioned in section 4.1.1. The process of capturing packets is also known as sniffing or snooping.

Snooping ARP Packets Snooping *Address Resolution Protocol* (ARP) packets, the need for a route discovery cycle can be identified. When a node does not know the physical address (MAC) of a host, an ARP request is used to discover physical addresses and map IP address to these [Com00]. When a node initially wants to communicate with another node, an ARP packet is sent to resolve the address. This approach was used in the Mad-hoc implementation of AODV protocol, which, however, is no longer maintained or available. The advantage of using this approach is that the implementation can be user space only.

One imminent problem with this approach is that ARP requests are only issued for hosts with addresses that either are a part of the subnet of one of the configured network interfaces or if a specific host entry exists in the routing table. The IP layer discards packets to destinations that do not match one of these entries. This is a problem as it requires nodes to be configured with addresses within the same subnet and prohibits nodes with unrelated IP addresses to join the network.

Additional disadvantages have already been explored by other sources [Wes03, KZG03, CJWK02] and are restated here for completeness. First, the approach requires complete management of the ARP cache to reduce risk of spurious requests as ARP requests are generated periodically depending on the status of the entries in the ARP cache [Bro03]. Similarly, although not likely to be a regular occurrence, if an entry is added manually to the ARP cache no ARP request is generated for the destination. Second, an implementation may have limited packet buffering functionality as it depends on the ARP implementation to queue packets and ARP implementations may buffer only one packet while ARP resolution is ongoing. Finally, packets are only buffered while ARP resolution has not timed out. The ARP timeout may occur within a relatively short period and before route discovery can be completed.

Snooping Data Packets Another approach for using snooping to identify the need for route discovery is to set up the loopback interface as the default route, and let the reception of any packet with a destination address different from the loopback address trigger a route discovery. Assigning the loopback address as the default route allows for easy buffering of outstanding packets; packets awaiting the completion of route discovery is queued in user space and inserted or discarded afterwards. The Airnet and JAdhoc AODV implementations use this approach [Les, KU03].

Snooping data packets going in and out of the monitored interface, route timeouts for active routes can be maintained and packets for hosts for which we have no route can be determined.

As mentioned previously, the Mad-hoc implementation uses the ARP snooping approach to identify the need for route discovery and snoops incoming and outgoing data packets to do its other housekeeping. JAdhoc and Airnet sets up the loopback interface as the default route and interprets any packets with destination different from the loopback address as an address for which route discovery should

be initiated. These implementations also snoop data packets as part of the other routing protocol operations.

In the following when we talk about snooping, we do not mean snooping of ARP request but only of data packets. The strength of the snooping strategy is that is possible to make an implementation solely in user space. This has two advantages. The first is that it allows for simple installation and also relatively simple execution. For example, no communication is required between user space and kernel space as with implementations that also require code to run in kernel space. The second advantage is portability. A user space only implementation using snooping is much easier to port to different platforms. The primary requirement is that the packet capture mechanism used is available on the other platforms. This is the case for the JAdhoc implementation, which is available for both Linux and Windows. It uses the *jpcap* network capture library [CKG+], which encapsulates packet capturing support for both Windows and POSIX operating systems. The Airnet [Les] implementation is only available for Linux, but the platform specific part of the code is isolated with the intent of easing porting.

The biggest weakness of the snooping approach is the overhead associated with the approach, as packets must travel to user space to be inspected.

4.2.3 Netfilter

Netfilter is a packet filtering framework implemented as a set of *hooks* at well defined places in the Linux TCP/IP networking stack. The netfilter framework allows user defined functions to be inserted and called as packets traverse the stack. Figure 4.1 gives an overview of the hooks defined for IPv4. Their location within the stack is shown with boxes, and illustrates when callback functions are called as packets traverse the stack. Functions are called on a per IP packet basis.

The upper part of the figure shows the two hooks NF_IP_LOCAL_OUT and NF_IP_LOCAL_IN that are traversed by packets either just after being sent or just before being received by a local process. At the lower part of the figure, the NF_IP_PRE_ROUTING and NF_IP_POST_ROUTING hooks are shown.

Incoming packets received on network interfaces traverse the NF_IP_PRE_ROUTING hook and packets to be sent by this host traverses NF_IP_POST_ROUTING just before being handed over the network interface driver. Thus, packets going from and to other hosts can be captured at these two hooks.

Routing decisions are made for packets arriving at the network interface of the host after traversing NF_IP_PRE_ROUTING, to see if they are bound for this host or destined to be forwarded. Routing decisions are made for packets sent by local processes after traversing NF_IP_POST_ROUTING.

Besides the above described pair of hooks, packets that are to be forwarded will in addition traverse the NF_IP_FORWARD hook.

Callback functions defined are inserted in form of kernel modules. Any registered functions for a hook must return a *verdict* for each processed packets telling netfilter what to do with the packet. The defined verdicts are as follows [Rus]:

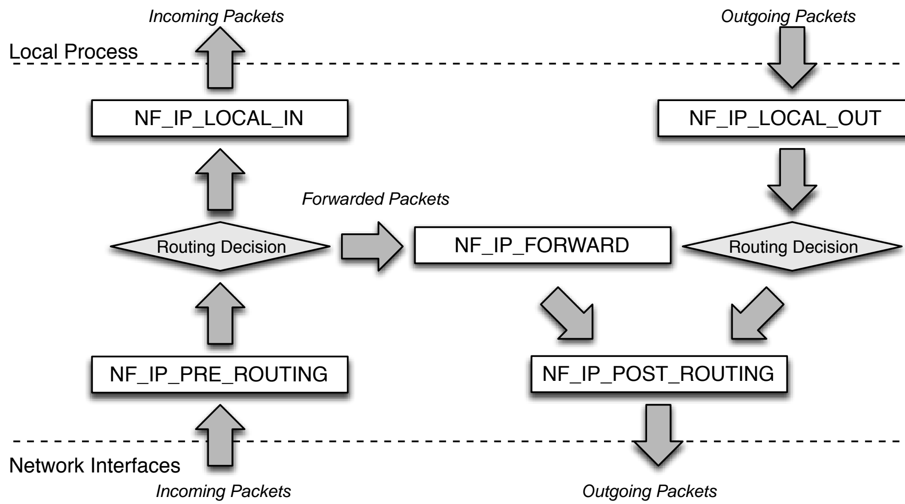


Figure 4.1: The netlink packet filtering framework [CBR04]. Arrows show the direction packets travel through the network stack as they enter from a local process or a network interface. The boxes show the possible hooks.

1. NF_ACCEPT: continue traversal as normal
2. NF_DROP: drop the packet
3. NF_QUEUE: queue the packet with the intent of reinjecting it later. A queue handler must be registered, otherwise this verdict is equal to NF_DROP
4. NF_STOLEN: take over control of packet. We may reinsert the packet at a later point in time
5. NF_REPEAT: call this hook again

In order not to corrupt the netfilter framework accounting, one must reinject any queued packets and return a verdict when finished processing them, i.e., it is not allowed to merely free the kernel memory allocated for packets.

Along with the netfilter framework is supplied a kernel module driver, *ip_queue*, and an accompanying user space library, *libipq*, providing packet buffering for user space. The netfilter framework provides excellent means to help in the implementation of on-demand ad hoc routing protocols and the challenges identified in section 4.1.1 can be met in the following way.

Registering a callback function at the NF_IP_LOCAL_OUT hook, packets can be captured before any routing decisions are made. If no route for the destination exists, route discovery is initiated and the packets can be queued for later reinsertion in the stack when route discovery completes. Packets can be queued either in the kernel or in user space. For further reference, we refer to these solutions as

the *netfilter kernel space packet buffering solution* and *netfilter user space packet buffering solution*, respectively.

At the NF_IP_POST_ROUTING hook, a callback function can be registered to record all sent packets. This way route timeouts for the destination can be updated both in the case when this host is the sender and when it forwards packets.

At the NF_IP_PRE_ROUTING hook, a function can observe whether a route for the destination of a packet exists when the packet traverses the hook. If not, an RERR can be issued. If a route does exists, the route timeout of the source node route can be updated. It is important that the check for active routes happens here, before any routing decision are made, as the IP layer would otherwise discard any packets with no active route making it impossible to have RERR messages issued.

4.2.4 Additional Implementation Issues

In the previous sections, the three primary methods for addressing the challenges outlined in section 4.1.1 for on-demand ad hoc routing protocol implementations were described. However, some issues and solutions have not been addressed yet, such as other ways to buffer packets awaiting route acquisition, the trade-offs between user space and kernel space implementations, and finally issues of letting packet forwarding interact with packet routing in user space.

Packet Buffering In section 4.2.3, it was mentioned that netfilter provides packet buffering capabilities. It is, however, not a requirement to use this facility in order to meet challenges 1 and 2 mentioned in section 4.1.1 and still use netfilter to support challenges 3 and 4.

The *Ad hoc Support Library* (ASL) [KZG03] uses a different approach to identify the need for route discovery and simultaneously buffer outstanding packets. ASL creates a local tunnel device, a virtual network interface, and points the default route to this device. The effect is that every packet for destinations with no route is sent to the device and is now available to a user space program. Whenever a packet is received on this device, route discovery can be initiated, and the packet can be buffered in a user space buffer. Netfilter is only used to update time stamp values for active routes. Because of a constraint imposed by netfilter, packets that are to be reinserted cannot be written back on the virtual network device they were received on. Instead, they are reinserted in the stack using a raw IP socket. A raw socket allows access to packet headers and packets written to a raw socket, thus to bypass parts of the network stack normally travelled by packets sent on normal sockets.

This approach is similar to *netfilter user space packet buffering solution* in which the netfilter packet queue facility is used to send outstanding packets for buffering in user space. The only visible difference is that the virtual tunnel device and the configured default route are visible if a user inspects the network interface configuration and routing table.

User Space vs. Kernel Space Implementations An on-demand ad hoc routing protocol can both be implemented solely in kernel space, solely in user space, or using a hybrid approach. With regards to a kernel-only implementation, one can use both the kernel modification or netfilter solutions. The whole motivation behind the snooping approach is to avoid the dependency on kernel level code and this technique is thus not applicable in this case.

The packet routing required in an ad hoc routing protocol involves complex CPU and memory intensive tasks, which are properly situated outside the kernel. If a single error occurs in the code running in the kernel, the whole system typically crashes. If an error occurs in a user space program, it stops working and we can actively terminate it, if it has not itself ended execution because of the error. The rest of the system continues execution with no errors.

There is also the issue of portability. A kernel level implementation is much harder to port to a different platform without requiring a complete rewrite. In user space, the implementation can rely on standard APIs while the kernel level programming environment typically varies widely between platforms.

One advantage of an implementation running only in kernel space is performance. No packets or control messages need to travel between user space and kernel space.

Kernel-AODV and NIST DYMO [KBb, KBa] are two kernel only implementation of respectively the AODV and DYMO routing protocols. Both implementations use the netfilter framework.

Forwarding Design As mentioned in section 4.1, the routing functionality in current operating systems is separated between the packet-forwarding function and the packet-routing function. The first function directs packets to the correct interface or local process based on the routing table and the second function maintains the actual operating system routing table.

Two designs using netfilter, meeting challenges 1 and 2 of section 4.1.1 was mentioned in section 4.2.3. Here we briefly present a third one, mixing the packet-forwarding and packet-routing functions.

Instead of only queueing packets to destinations that have no valid route, every packet is sent to the user space daemon. For each packet crossing the kernel/user space boundary, a forwarding decision is made by the ad hoc routing protocol daemon in user space. Packets are matched against a routing cache maintained by the daemon and are either queued if a route discovery cycle must be initiated or returned to the kernel immediately if a route exists. The daemon thus takes the responsibility of both the packet-forwarding function and the packet-routing function.

The advantage using this approach is that the amount of code running in the kernel needed to support the implementation is small. The major drawback is the inefficiency of this approach: every packet is copied from kernel space to user space and back again. Studies show packet processing times to be ten times longer

when compared to the kernel design [CBR05]. In addition, routing is done twice. Once by the daemon in user space and once again by the kernel.

The AODV-UCSB implementation [CBR04] and the AODV-UU [Nor] implementation prior to version 0.9 used this approach. Since version 0.9, AODV-UU has used the *netfilter kernel space packet buffering solution* described in section 4.2.3.

5 DYMO-AU Design and Implementation Overview

The following two chapters describe the design and implementation of the DYMO routing protocol for Linux. The implementation is called DYMO-AU. In this chapter, we give an outline of the design and implementation of DYMO-AU. In the next chapter, we go into further details about the design and implementation of DYMO-AU. When pertinent, we use message diagrams [Fow03] to illustrate the dynamic interaction between the different parts of the implementation during protocol operation.

In continuation of the discussion of the different implementation approaches in section 4.2, we present the chosen implementation approach in section 5.1. In section 5.2, we give a short overview of the structure of the implementation and introduce the various parts and modules the implementation consists of. In section 5.3, we introduce the Lua programming language which has been used in our implementation. In section 5.4, we present the communication interface that allows information exchange between user space and kernel space. Finally, in section 5.5, we discuss errors in the DYMO specification, how our implementation deviates from the DYMO specification, and portability of our implementation.

5.1 Design Approach

In section 4.2, we presented various implementation techniques for implementing on-demand routing protocols on Linux. In the DYMO-AU implementation, we have chosen to use the netfilter framework and to implement the DYMO protocol as a user space routing daemon with an accompanying kernel module.

Because of the salient disadvantages of the kernel modification approach, we did not consider this technique. Compared to the snooping approach, with the netfilter approach one avoids the overhead of copying data packets to and from user space. However, we also must abandon a user space only implementation. Although, a Linux kernel module makes the implementation more complex, it also

adds flexibility. For example, the update of route timeouts can be implemented in various ways, as we describe in section 6.3.

5.2 Implementation Overview

The DYMO protocol is implemented as a user space daemon written in C and in the scripting language Lua [IdFC03, Ier03] (see section 5.3 for a short introduction), with an accompanying Linux kernel module.

In the user space daemon, all protocol logic of DYMO is written in Lua and all network code is written in C. The kernel module is used to detect the need for route discovery, updating of route timeouts when sending, receiving and forwarding packets, and queueing of packets for which route discovery is in progress. The user space daemon communicates with the kernel module using Linux *netlink* sockets [He05, Net99].

The design of DYMO-AU and its two major components, the routing daemon and the kernel module, is illustrated in figure 5.1. As mentioned above, the user space routing daemon consists of a module handling protocol logic and a module handling network I/O including event dispatching. The protocol logic module can be further divided into sub-modules, and the inner workings of these modules are described in section 6.1.

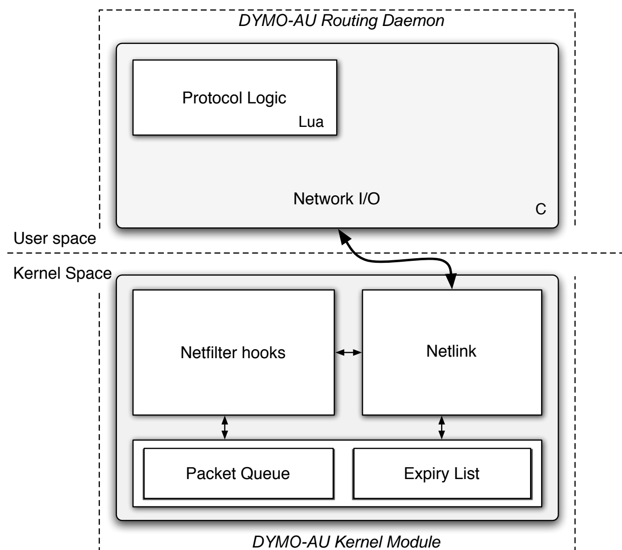


Figure 5.1: The components of DYMO-AU. In user space, the implementation consists of a routing daemon written in C and Lua. In kernel space, the implementation consists of Linux kernel module that is made up of four parts.

The Linux kernel module consists of four different components. They are depicted layered based on their dependencies as will be explained below. The division of the kernel module into four parts is inspired by the design of the AODV-

UU [Nor] implementation. In the following, we shortly introduce the four components of the kernel module.

5.2.1 Packet Queue

The packet queue module at the lower left implements a queue used for buffering packets awaiting the completion of a route discovery.

5.2.2 Expiry List

The expiry list module at the lower right is used to track usage of routes. Whenever a route is used, the accompanying time stamp associated with the route is updated to the current time. The life span of entries in this list can either be controlled from user space or using a hybrid approach, adding the entry for a destination from user space and deleting it from kernel space. We discuss the possible solutions and associated advantages and disadvantages in section 6.3.

5.2.3 Netlink Communication

The netlink module is responsible for the communication between user space and kernel space. The communication is implemented using Linux *netlink* sockets. The DYMO protocol events are exchanged between the user space daemon and the kernel module using a netlink socket. The communication channel is bidirectional and the user space process is not restricted to initiate transfers. In the figure, the module is layered on top of the packet queue and expiry list, meaning that these are manipulated when control messages from user space are received. We give the full details of the netlink component in section 6.2.3 and the details of the interface between user space and kernel space in section 5.4.

5.2.4 Netfilter Hooks

The netfilter hooks module generates the events that trigger routing protocol action. This is implemented as a function that hooks into three different places within netfilter. Matching packets sent from a local process against the expiry list tells the module if a route for the destination is known. If no route is known, the module tells the user space daemon to initiate the route discovery process for the destination if it has not done so already. Querying the packet queue provides the latter information. When receiving packets on a network interface, we can similarly ensure that we do not receive packets for unknown destinations or for destinations which route table entry has timed out. In the event of receiving such unsolicited packets, a route error event is generated and sent to the user space daemon. We elaborate on the design of the kernel module and choices made regarding the design in section 6.2.

5.3 The Lua Programming Language

In this section, we briefly introduce the Lua programming language [IdFC03, Ier03] and its implementation, and why we used it in our implementation to write the modules that implement the protocol logic of DYMO.

Lua is a procedural scripting language designed specifically for extending applications written in other languages, primarily C and C++, but an implementation for Java [Pro], as well as bridges to Python [Nie] and Objective-C [Bal] also exists. Lua is dynamically typed, interpreted from bytecode, and memory is automatically managed using garbage collection.

Lua has a fast and light-weight implementation. It is light-weight in the sense that it has a low embedding cost, adding between 60 KB and 150 KB to the size of the host program. Lua is fast in the realm of scripting languages as implementations for dynamically typed, interpreted languages cannot be compared to languages with implementations producing natively compiled binaries. Independent benchmarks show it to be faster than implementations of, for instance, Perl, Python, PHP, Ruby, and Javascript [Sho, Byt]. Beside the standard Lua virtual machine implementation, there exists a just-in-time (JIT) compiler framework for Lua called LuaJIT [Pal06].

Although Lua features procedural syntax, it can be described as a multi-paradigm language. Its functions are first-class values and have proper lexical scoping. This allows programs to be written using techniques from the world of functional programming. Lua provides meta-mechanisms to implement additional features as well as to modify the existing ones. The meta-mechanisms in conjunction with first-class functions leverages the support for object-oriented programming. The support for object-oriented programming has been used in the implementation of DYMO-AU.

One important feature of Lua that has been used in the DYMO-AU implementation, is the ability to extend Lua with user-defined types in C. Using this functionality, it is possible in Lua to manipulate objects defined and created in C as well as creating objects in Lua that can be handed over to C. To ease the integration between C and Lua, the tool *tolua++* [Manb] has been used to automatically generate the glue code that allows interoperability between the parts of DYMO-AU that are written in C and the parts written in Lua.

We chose Lua to speed up and ease the implementation process. For example, while working with our implementation, the DYMO specification was subject to a major change when the draft was updated from version 3 to version 4. Because of the higher-level nature of the language, we were able to adopt the code to the new version quite easily. The Lua implementation has support for dynamical compilation and execution of source code entered as a string. This facility makes it easy to support inspection of data structures using the standard functions of the Lua language. To aid with debugging and testing during the development of the DYMO-AU implementation and when performing the experimental evaluation, we used the support for dynamical execution to allow inspecting the data structures of

a running instance of the routing daemon. Lua is interpreted and if performance should become a problem, it is easy to incrementally improve execution time, by rewriting selected parts of the Lua code in C.

5.4 User Space-Kernel Space Interaction

This section gives an overview of the communication that takes place between the routing daemon and the kernel module. We describe the messages that have been defined and how these are transmitted between the two layers during protocol operations.

5.4.1 Message Types

The messages that have been defined for user space and kernel space interaction is described in the following. To give a better overview, the next action performed by the recipient of the message is also described.

NO_ROUTE This message is sent from the kernel module to the user space daemon when a user space application tries to send packets to a destination for which no route currently exists. The user space daemon then initiates route discovery. This only happens at a node originating traffic.

The message is also sent from the user space daemon to the kernel module after unsuccessful route discovery. The kernel module then drops the queued packets for the particular destination.

ADD_ROUTE Sent from the user space daemon after successful completion of route discovery. Tells the kernel module to install a route to the destination and send all buffered packets.

DEL_ROUTE Sent from the user space daemon. It informs the kernel module to delete a route for a destination, when the user space daemon detects that it has become stale.

RERR_IN Sent from the kernel module when a packet is received for a destination with no route table entry. The user space daemon generates and transmits an RERR message for the given destination.

PKT_UPDATE_INBOUND An incoming packet arrived. Tells the user space daemon to update the route timeout for the source address of the packet.

PKT_UPDATE_OUTBOUND A packet was sent on an outgoing interface. The user space daemon updates the route timeout for the destination address of the packet.

Both update messages are sent to avoid route timeouts for active routes. Although, we currently do not take advantage of it, two different messages exist to be able to distinguish the two events in case separate actions are required.

5.4.2 Communication Interface in the Daemon

Seen from the Lua code in the routing daemon, the communication interface is platform independent. A special *kernel communication* interface is defined and exposed to Lua. Thus, nothing special about netlink sockets and netlink messages are exposed in the code. The Lua routing logic module only knows that it is possible to send the above described types of messages to the kernel and the name of a function for sending them. The goal is to allow for greater portability, i.e., to use another message passing technique under the hood if the implementation is ported to other operating systems.

5.4.3 Route Discovery Example

We now give two examples highlighting the interaction between kernel space and user space. Figure 5.2 shows the interaction that takes place between the routing daemon and the kernel module during successful routing discovery. As emphasis is on the communication interface and the overall picture, the operations performed in both the routing daemon and kernel module are abbreviated.

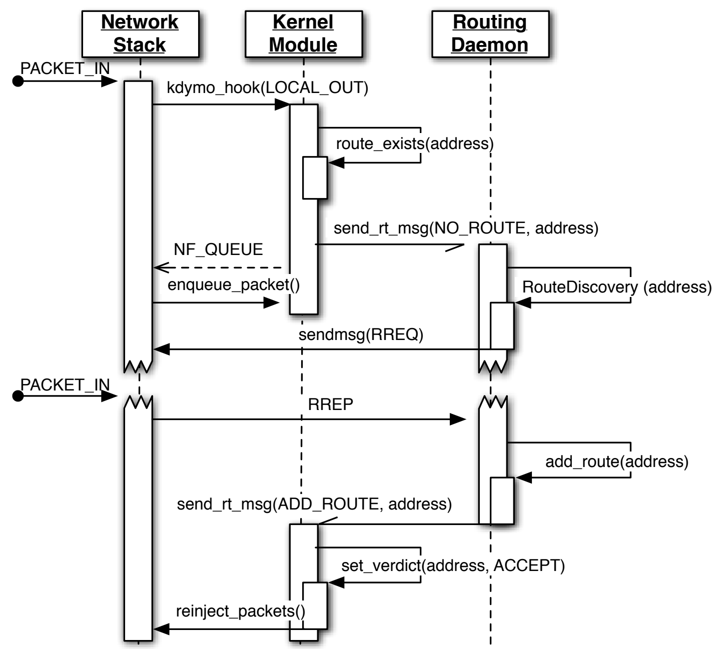


Figure 5.2: A route discovery example. As messages between daemon and kernel module are sent asynchronously, work is being shown as performed simultaneously in the two layers after the NO_ROUTE message has been sent.

In figure 5.2, the function registered with netfilter by the kernel module is called when a local application tries to send a packet. This is shown with the message labelled PACKET_IN going into the Network Stack column, which then calls the

kernel module. The kernel module checks if a route for the destination of the packet exists. If no route exists, a NO_ROUTE message is sent to the routing daemon telling that a packet was sent by a local application for a destination with no valid route. The kernel module returns NF_QUEUE (see section 4.2.3) and the kernel module packet queue is then called to buffer the packet. In user space, the routing daemon initiates route discovery and an RREQ is sent using broadcast. When the routing daemon receives the RREP, it is processed and the route for the destination is added. An ADD_ROUTE message is sent to the kernel module to add a route for the destination and reinject any buffered packets into the TCP/IP stack.

5.4.4 RERR Processing Example

The next example illustrates RERR processing. RERR processing is mostly done by the daemon in user space, however when routes are being invalidated, DEL_-ROUTE messages are sent to the kernel module to also remove routes there.

RERR processing has already been detailed in section 3.3, but in order to illustrate the example, the processing rules are summarized here. Figure 5.3 shows how RERR are processed when received by a node. Each of the unreachable nodes contained in the RERR is now tested the following way. If a route table entry exists for the node, it is invalidated if the following three conditions are met. It is invalidated by setting the valid timeout to the current time which means the entry will eventually be deleted. In addition, a DEL_ROUTE message is sent to the kernel module to delete the entry for the destination from the expiry list. The three conditions are:

1. The next hop address of the entry is the same as the IP source address of the RERR.
2. The next hop interface of the entry is the same as the one the RERR was received on.
3. The sequence number for the unreachable node is unknown or it is less than or equal to the sequence number of the corresponding route table entry.

The two first conditions must be met to ensure that entries we are invalidating have the node that sent the RERR as next hop entry, i.e., the source of the RERR is on the path to an unreachable node. The third condition ensures that only fresh routes are invalidated to protect against propagating old information.

5.5 Discussion

In this section, we cover errors found in the DYMO specification during the implementation process. We cover features missing in the DYMO-AU implementation in order to comply with the DYMO specification. Finally, we cover portability of the DYMO-AU implementation.

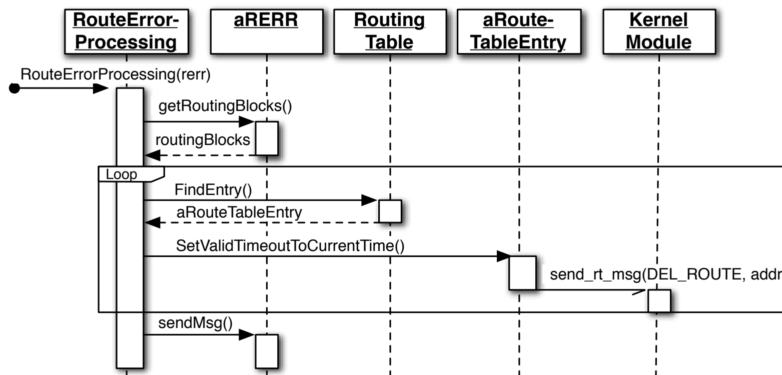


Figure 5.3: RERR processing. The loop is executed for each route table entry that must be invalidated. For each of the entries, a DEL_ROUTE message is sent to the kernel module.

5.5.1 Errors in the DYMO Specification

In the fourth revision of the DYMO specification draft [CP06b], the section on creating or updating a route table entry from routing message (section 4.2.1 in the specification), lacks a paragraph on route staleness when the sequence number for a destination found in the routing message is greater than the one in the routing table (when Node.SeqNum - Route.SeqNum is greater than 0). This paragraph can be found in the third revision as well deduced from the fifth (the structure and phrasing of the text in the fifth revision has been rewritten compared to earlier revisions). As this is an obvious error in the specification, the DYMO-AU implementation implements the behaviour from the third and fifth revision.

In section 4.3.2 (of the DYMO specification [CP06b]) on routing message processing, the phrasing of how the hop count value for entries in the message should be updated is ambiguous:

> For each of these addresses the Node.HopCnt associated with the address is incremented by one (1) if it exists and is not zero, then a route is created or updated as defined in Section 4.2.1. The updating of the HopCnt occurs after processing.

One might wonder whether the hop count should be updated before or after, routes are created or updated. The specification says that if a node appends information about itself when forwarding a routing message, the hop count value should be set to one. With this in mind, we conclude that the hop count should be processed *after* processing, because if the hop count is incremented before processing, a neighbour receiving the message would record a hop count of two to the node. One of the authors of DYMO, Ian Chakeres has confirmed this conclusion when inquired in an e-mail.

A similar flaw was found in the fifth revision of the DYMO specification draft and Ian Chakeres once again acknowledged the error. The error has been corrected in the sixth version of the draft.

5.5.2 Limitations of the DYMO-AU implementation

In this section, we describe the limitations of the DYMO-AU implementation compared to the DYMO specification [CP06b] as well as the ways it deviates. The DYMO-AU implementation has the following limitations when compared to the specification.

- No support for active link sensing (HELLO messages, Link layer feedback, Neighbour discovery); only route timeouts are supported.
- No gateway support.
- Crude support for the generalized MANET packet/message format limiting address prefixes to 24, i.e., netmask 255.255.255.0.
- Packet generation limits not supported.
- Actions after sequence number loss not supported.
- Only one interface supported (much of the code needed has already been developed, but it has not been enabled or tested yet).
- As the specification does not define the MANETcast address, the configured broadcast address on DYMO enabled interfaces is used.
- Only IPv4 supported.

In section 6.3.1, we mention that when the route timeout for a route table entry is updated to the current time, the next hop entry of the route table entry is also updated. Updating of the next hop entry is not mentioned in the specification, but is required in our implementation to avoid the entry for the next hop node to time out, because link sensing is not implemented.

5.5.3 Portability

Currently, the DYMO-AU implementation only supports Linux, but it has been designed with portability in mind. For instance, as described in section 5.4.2 the kernel communication interface is not Linux specific. The approach we have used to ensure portability is called the *intersection* approach. It is discussed by Kernighan and Pike in their book *The Practice of Programming* [KP99] along with another approach called the *union* approach.

The intersection approach seeks to only use the features available on all target systems. In practice, this is realized by hiding system dependencies behind interfaces. If two systems provide two different ways of accomplishing the same thing,

a portable interface is implemented and the system dependencies are localized in separate files. The implementation required to adapt to a given system might vary widely from other systems.

The union approach is to use the best features of each particular system, and make the compilation and installation process conditional on properties of the local environment. This is typically accomplished in C and C++ by using the preprocessor macros, e.g., `#ifdef` and `#define`, and make the code conditionally depend on the properties of the system that the code is compiled on.

As discussed by Kernighan and Pike, the intersection approach is often preferable as the union approach makes the source code hard to read because of the required conditional compilation preprocessor macros. This is especially true when compile-time control flow defined with preprocessor macros is mixed with run-time control flow (if-statements). The union approach can be acceptable if used occasionally and can sometimes be favourable in simple cases if the unreadability caused by using conditional compilation to alter code to adapt to a new system is tolerable, compared with the extra work required to build a portable library.

6 DYMO-AU Design and Implementation Details

In the previous chapter, we gave an overview of the DYMO-AU implementation. In this chapter, we go into details about the design and implementation of DYMO-AU.

In section 6.1 we give the full details of the parts of the implementation that run in user space and then in section 6.2 we give the full details of the kernel space implementation. In section 6.3, different strategies for updating route timeouts in the routing daemon is described.

6.1 The User Space Routing Daemon

In this section, we give an overview of design and implementation of the routing daemon and the modules comprising the daemon.

As mentioned in section 5.2, the user space daemon is implemented in C and Lua. An overview of the different modules is given in figure 6.1.

The modules depicted in the figure are described in the following. For now we just note that any ingoing traffic, visualized with the arrow labelled with **Input**, is monitored using the `select` [SFR03] system call and upon detecting input, the function registered for the associated socket is called. As a result, some protocol processing follows and possibly a DYMO message is created and sent from the protocol logic module, visualized with an arrow labelled **Output** going out from the Protocol Logic module.

6.1.1 Timer Queue

A DYMO routing daemon must ensure that invalid route table entries are deleted after their delete timeout has passed. This requires us to be able to execute a task at a preset time in the future. To support this purpose, a Timer Queue module has been implemented. To have some task executed after a certain time interval has

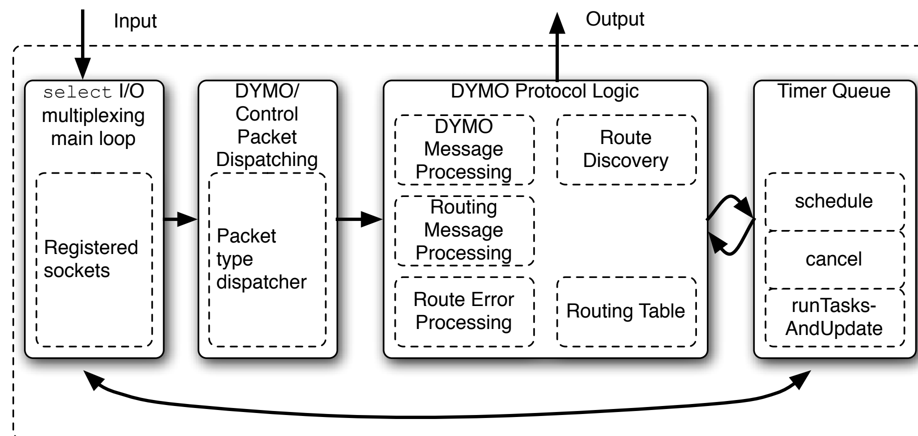


Figure 6.1: DYMO-AU routing daemon module overview.

elapsed, one creates an object of type `TimerTask`. The task to be executed must be defined in the form of a function. Timer task objects can then be scheduled with the timer queue and in addition, be cancelled if needed. The timer queue internally uses a priority queue ordered according to the timeout value of the timer tasks in the queue. The timer task with the earliest timeout value, i.e., the timeout value that comes before the timeout values of all the other tasks, is the element returned when retrieving the head element of the queue. The time stamp for timer tasks is expressed as an absolute value, i.e., it is obtained using `gettimeofday`, which returns the time since midnight, January 1, 1970. The desired time interval is then added to this value.

The timer queue does not itself make sure that timer tasks are executed at appropriate intervals, but instead provides a function which when called, executes any timer tasks that have timed out, i.e., whose timeout value is before the current time. When all of these tasks have been run, the timeout value of the timer task at the head of the queue is returned. This task is the one with the timeout value closest to the current time while still in the future.

The intentional uses of the timer queue function is in conjunction with, for instance, the POSIX synchronous I/O multiplexing system functions `poll` and `select` [SFR03], which both take a timeout value as argument. In the DYMO-AU implementation, `select` has been used (see section 6.1.2 for further details). Whenever `select` returns it is because one of two things occurred:

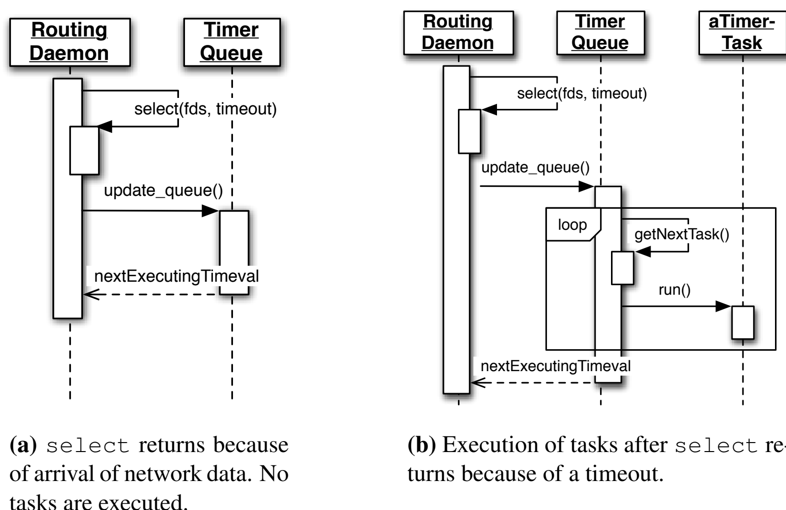
1. Data is ready to be read from one of the supplied file descriptors.

2. The call to `select` timed out.

The two scenarios are illustrated in figure 6.2. In the first case, figure 6.2a, when the timer queue update function is called after processing the ready descrip-

tors, no task is ready to execute: A new timeout value (giving the new timeout interval, labelled *nextExecutingTimeval*) in the figure is returned to be used for a new call to `select`.

In the second case, figure 6.2b, `select` returns because it timed out. When the timer queue update function is called, at least one task is guaranteed to execute as the timeout value of this task is the one returned by the previous call to the update function. Tasks in the queue are removed and executed as long as their timeout value is before the current time. As in the first scenario, the queue eventually returns the next timeout interval.



(a) `select` returns because of arrival of network data. No tasks are executed.

(b) Execution of tasks after `select` returns because of a timeout.

Figure 6.2: Timer queue update function examples. In both cases the nextExecutingTimeval value returned, is the next timeout interval of the timer queue.

The design outlined above has been chosen as it, compared with an implementation where the timer queue runs within its own thread of execution using POSIX threads [MS04], makes the code less complex and easier to understand. Multithreaded programming is inherently complex and makes it difficult to locate errors. Interoperability between `select` and the timer queue is described further in the next section.

6.1.2 `select` I/O Multiplexing Main Loop

The routing daemon must be able to listen for data on more than one socket, e.g., both on a socket listening on the DYMO port and on the netlink socket used for communicating with the kernel module. Therefore, one cannot call, for instance, `recv` [SFR03] on an open socket as the call would block until a message is received on the socket. Other open sockets would be prohibited from receiving mes-

sages without the use of threads. In addition, if calling `recv`, it would be difficult to have tasks executed at a preset time in the future without the use of threads.

The `select` call makes it possible to listen for messages on multiple socket descriptors and in addition, to supply a timeout value that specifies for how long the call should block waiting for data. These possibilities are used by the routing daemon, which was also depicted in figure 6.1. Here any registered sockets are a part of the watched set of descriptors. The timeout value to use is obtained from the timer queue module. To ensure correct interoperability between the timer queue and `select`, the timeout value obtained from the timer queue must be converted, as `select` expects the timeout as an interval. Consequently, the timeout value obtained from the timer queue is converted by subtracting the current time from the value.

Figure 6.3 gives a different view of the interaction between the main loop and `select`, the timer module, and packet dispatching, which is described in the next section. The `select` call watches input from two different sources: the DYMO routing socket, here assigned port number 13430, and the netlink socket for exchanging control messages between the routing daemon and the kernel module. The port number 13430 was chosen arbitrarily for the DYMO-AU implementation, as no port number has yet been assigned to the DYMO protocol.

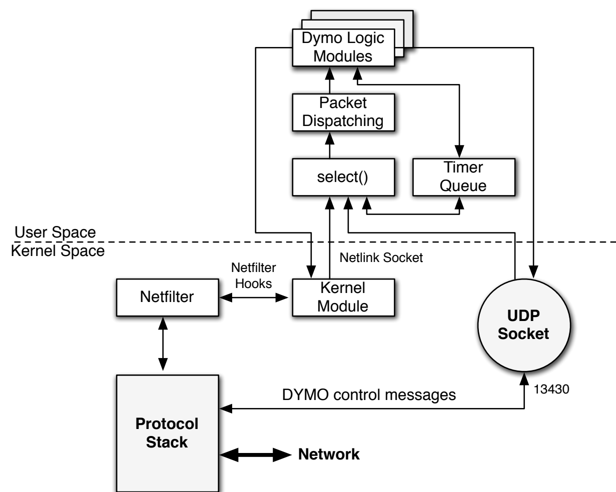


Figure 6.3: Selecting from multiple sockets and providing timed events [Wib02].

6.1.3 DYMO and Control Packet Dispatching

The DYMO and Control Packet Dispatching module, shown to the left of the main loop module in figure 6.1, is responsible for handling DYMO messages and also control messages from the kernel module (e.g., instructing the routing daemon to

initiate route discovery) and take appropriate action according to the type of message received. The code for processing DYMO messages and control messages, respectively, is in separate source code files so when we talk about a common module, it is because of the conceptual similarities between the two pieces of code.

As described in section 5.2, the network code of the routing daemon is written in C and consequently when a DYMO message is received, it is first handled in the C code. The C code reads the packet type from the message header and for each of the different types it calls a function defined in Lua to proceed with the processing defined for the individual packet types.

As is the case with the DYMO messages, the code that receive kernel control messages is written in C. Depending on the type of message received, the C code calls the appropriate Lua function.

6.1.4 DYMO Message Processing

The module responsible for processing of messages parses the content of DYMO messages according to the generalized MANET packet and message format described in section 3.4. A consequence of the flexibility of message layout is that one can generally not be sure to find a specific field at a fixed offset within a message. The content of a message must be parsed, extracted, and represented in a form so the daemon can process and use the embedded information.

Each address and associated attributes, i.e., prefix value, hop count, gateway bit, and sequence number, are extracted and represented in a data structure that we refer to as a routing block. The routing block term was also used in the DYMO specification prior to version 4 of the draft. All further processing of DYMO messages happens on this list of routing blocks. This has the added advantage that the routing logic is independent of the actual binary representation of DYMO messages. The continued processing of messages happens in the Routing Message Processing (section 6.1.5) and Route Error Processing (section 6.1.7) modules.

6.1.5 Routing Message Processing

This module handles routing message processing as specified by the DYMO specification. RREQ and RREP messages contain the same information but have slightly different processing rules. Consequently, all routing messages are processed by the same function. The function performs the same overall processing on messages, but takes specific measures when the packet type being examined requires it.

The processing common for routing messages includes creating and updating of route table entries as the routing blocks in the message is processed. If a route table entry is created because of processing the current routing block, a message is sent to the kernel module telling it to add an entry to the expiry list. A similar message is sent if an entry requires update.

6.1.6 Route Discovery

This module handles the route discovery process. Whenever a route to a node is needed, and it has not previously been discovered or only a stale route table entry exists, a route discovery cycle is started. An RREQ message is created and sent on the interface configured for DYMO operation. A timer task is then created to allow for repeated route discovery attempts if an RREP is not received within RREQ_WAIT_TIME (see section 3.2). Information about the currently ongoing discovery attempt is then inserted into a table with the destination address as the key and the discovery information including the created timer task as value.

If an RREP is received for an address for which route discovery is in progress, the scheduled route discovery timer task for this address can be retrieved from the table of currently ongoing discovery attempts. The timer task is cancelled and a message is sent to the kernel module to inform that a new route has been created and to send any queued packets. Because of how routing messages are being processed, the route has already been added during routing message processing as described in section 6.1.5.

If no RREP is received within RREQ_WAIT_TIME, the created timer task executes and a new RREQ is created and transmitted. This happens a total of RREQ_TRIES (3) times and each time the waiting time before sending another RREQ is doubled. If no RREP messages are ever received, a message is sent to the kernel module to drop any buffered packets that have been queued while awaiting the completions of the route discovery cycle. The details of the working of the kernel module is given in section 6.2. A diagram showing the interaction between the kernel module and the routing daemon when routing discovery is carried out can be found in figure 5.2

6.1.7 Route Error Processing

This module handles routing message processing as well as the creation of RERR messages. Figure 6.4 illustrates the RERR generation process in the routing daemon. RERR messages are generated when a packet for a destination without a valid route table entry is received. When receiving a notification about an unreachable node, an RERR message is created and information about the node is added to the message. Additional unreachable nodes dependent on the same link, i.e., nodes having the unreachable link as next hop entry, may be added to the RERR. The RERR message is then broadcasted.

An example showing processing of RERR messages, focusing on the interaction between the user space routing daemon and the kernel module was previously given in section 5.4.4.

6.1.8 Routing Table

The DYMO specification describes a conceptual data structure, the route table entry, with fields used to ensure correct functioning of the DYMO routing protocol.

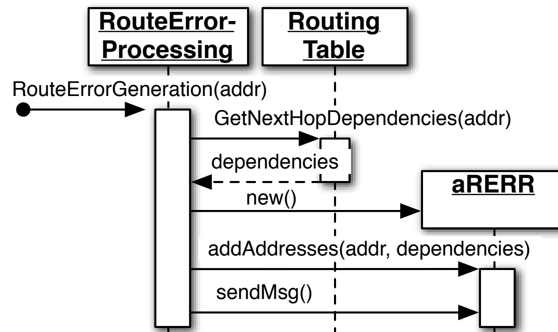


Figure 6.4: RERR creation.

We described the route table entry in section 3.1. This data structure together with operations to operate on a collection of entries, is implemented in the Routing Table module.

The routing daemon maintains the routing table in user space with the intent of making message processing as efficient as possible; we want to avoid querying the kernel routing table for each processed routing block in the message. Furthermore, additional information about nodes is needed, e.g., the sequence number, which it is not possible to store in the kernel routing table.

As mentioned in section 6.1.1, it is necessary for the correct function of the daemon that specific tasks can be executed at some point in the future. For instance, the daemon must ensure that route table entries are eventually deleted if they have not been used for some time. Accordingly, a timer task is created every time a new route table entry is created and scheduled to fire when the entry becomes invalid. As the entry has just been created this is equal to ROUTE_VALID_TIMEOUT milliseconds into the future. When the task eventually is executed, the function first checks if the entry is valid, as the valid timeout might have been updated in the meantime as a result of packets being transmitted using this route (we elaborate on this issue in section 6.3). What happens next depends on the state of the route table entry:

Valid If the route table entry is valid, the timer task is rescheduled to perform the check again, the next time it becomes invalid.

Invalid If invalid, the entry must be deleted eventually: a new timer task is scheduled, which will delete the entry ROUTE_DELETE_TIMEOUT milliseconds later.

To sum up: the timer task that checks the validity of route table entries can both reschedule itself or schedule a timer task that will delete the route table entry. Both cases are illustrated in figure 6.5: If the entry is still valid, the ValidTimeoutMonitor requests the value of the valid timeout from the route table entry and create a

new ValidTimeoutMonitor, which is subsequently scheduled. This means that each instance of *X* should be replaced by *Valid* in the figure. Similarly, if the route table entry is invalid, ValidTimeoutMonitor requests the delete timeout value from the route table entry, creates a DeleteTimeoutMonitor and schedules it. In this case, *X* should be substituted with *Delete* in the figure.

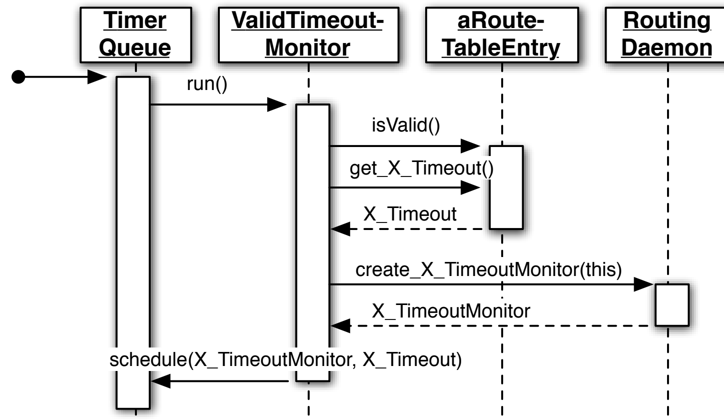


Figure 6.5: A timer task function checking route table entry validity. The type of timer task depends on the result of `isValid()`.

When a route table entry is initially created for a destination, a route is also added to the kernel routing table and similarly when the entry is deleted, the corresponding entry is deleted from the kernel routing table. In the following, we list three possible methods to alter the kernel routing table. The first method to manipulate the kernel routing table is with the use of *routing sockets* [SFR03]. A routing socket works very much like an ordinary socket; messages or commands can be written and responses read back. Linux however, does not support standard routing sockets, but has its own implementation called rtnetlink routing sockets or NETLINK_ROUTE routing sockets. The rtnetlink routing sockets provide a functionality super set of ordinary routing sockets [He05], but no compatibility layer between the two inferfaces exists, i.e., code using ordinary routing sockets does not compile on Linux.

A second approach is to use `ioctl` operations with the SIOCADDRT and SIOCDELRT requests. This works on Linux but is not guaranteed to work on other systems [SFR03].

A third option is to use already available utilities for manipulating the kernel routing table.

The two first options require the implementation to be written in C. If we should port the DYMO-AU implementation from Linux to another POSIX operating system, the kernel routing table manipulating code may have to be rewritten. Given these considerations and furthermore to avoid spending too much time writing low-

level kernel routing table manipulation code in C, we decided to use the already available routing table manipulation utilities and call these directly from Lua. The current Linux implementation uses the utility *ip*. The disadvantage to this approach is the added overhead; to call external commands, the Lua implementation uses the `system` function [KR88], which executes the command in a `fork`ed [MS04] process. To minimize this overhead, the daemon only deletes the kernel route table entry when the associated route table entry in the daemon is deleted, even though the route may be invalid.

In figure 6.6, the lifetimes of various types of route table entries are illustrated, among them kernel route table entries, which resemble the lifetime of the daemon entries. As a user space routing table entry can be made invalid and valid several times after it is created, the corresponding expiry list entry can be deleted and added several times (we describe the details of the expiry list in section 6.2.2). In figure 6.6, the cross symbolizes deletion of an entry.

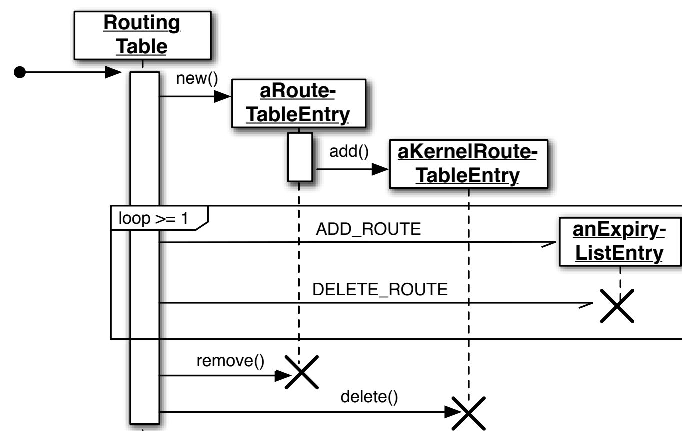


Figure 6.6: Lifetimes of various route table entries. A kernel route table entry has the same lifetime as a daemon entry whereas kernel expiry list entries can be added and deleted more than once.

6.2 The Kernel Module

In this section we describe the design and implementation of the components making up the Linux kernel module as depicted in figure 5.1.

6.2.1 Packet Queue

One of the requirements for an on-demand ad hoc routing protocol to function is that packets awaiting the completion of route discovery must be buffered. As mentioned in section 4.2.3, netfilter provides a packet queueing module *ip_queue*

and an accompanying user space library *libipq*. The ip_queue module implements the required functionality, but the interface to the kernel module and its user space control library has two limitations.

The first and most serious limitation, is that for the daemon to be able to obtain the destination address it would require that the whole IP packet is copied to user space. The second limitation is that libipq wraps the socket functions and `select` in its own API which makes it impossible to have it work directly with `select` without rewriting parts of libipq. For this reason, we decided to implement a separate packet queueing module. However, this module has been modelled after ip_queue.

The packet queue module is implemented as a netfilter queue handler. It must register itself with netfilter supplying a callback function with the prototype shown below.

```
nf_queue_handler(struct sk_buff *skb,
                 struct nf_info *info, void *data);
```

Whenever the verdict NF_QUEUE is returned from the netfilter hook function, this callback function is called with three arguments. The first two are a pointer to a `struct sk_buff`, which is the packet to queue, and pointer to a `struct nf_info` that provides auxiliary info about the packet, like which hook and interface the packet was on. When route discovery has finished, the kernel module is sent a message telling it to either send the queued packets or drop them. If a packet is dropped, an ICMP destination unreachable message [Com00] is created and delivered to the user space application that attempted to send the packet.

After the implementation of this module was finished it was discovered that it is not necessary to copy the whole packet to user space with ip_queue and libipq. Only a user specified part of the beginning of the packet has to be copied, for instance, just enough to be able to read the IP destination address from the packet. However, in order to use ip_queue and libipq, the above-mentioned incompatibility between the libibq API and `select` would still have to be resolved.

6.2.2 Expiry List

As described in section 5.2.2, the expiry list module is used to keep track of routes and their usage. It acts as the list of active routes for the kernel module, i.e., it only returns an entry if a valid route to the destination exists at time of inquiry. As described in section 6.1.8, the routing daemon also maintains a routing table, so one might ask why an additional table in the kernel is necessary.

When a host receives packets that must be forwarded instead of being delivered locally, it is desirable that the packets are processed as fast as possible. That is, when a route has been discovered by the daemon and is installed, minimal processing is desired.

If the implementation was made such that the kernel module had to query the user space daemon about every received packet, it would add additional latency

for each packet processed. This approach can roughly be compared to the one described in section 4.2.4, in which every packet is copied to user space. As described, experiments have shown the forwarding delay to be ten times longer when a packet must travel through user space [CBR05]. As we maintain the packet queue in the kernel, the amount of data transmitted through user space would be much smaller, but it is still anticipated that the added overhead of transmitting queries to user space would be significant.

As noted by Kawadia et al. [KZG03], freshness information about routes is recorded by the Linux kernel and can from kernel space be looked up in the kernel routing cache. However, as they also note, the time stamp is not readily available from user space even though the routing cache entries can be read through files found in the */proc file system*. The /proc file system is a virtual file system that let user processes access information on statistics, parameters, and data structures in the kernel as well as the ability to modify kernel variables by writing to selected files in the /proc file system hierarchy [MS04]. As it is desirable to be able to read these values from user space, the list of currently active routes is maintained in the kernel together with a time stamp indicating last usage. The reason we maintain a time stamp independently, rather than obtain it from the routing cache when requested, is that it is not desirable to depend on the lifetime of entries in the routing cache. Entries can be deleted while we are still interested in obtaining the time stamp. Situations where this might be the case is described in section 6.3.3.

To sum up, during normal mode of operations, the lifetime of an entry in the expiry list is controlled by the routing daemon. As described in section 6.1.5, expiry list entries are created when route table entries are created or updated, as a result of a routing message being processed by the daemon. When a route table entry becomes invalid, the daemon tells the kernel module to delete the expiry list entry. Whenever a route is used, the accompanying time stamp associated with the route is updated to the current time.

As an expansion of the normal mode of operations, entries might also be deleted by the kernel module itself. We give the full details in section 6.3, specifically section 6.3.3.

6.2.3 Netlink Communication

As described previously in section 5.2.3, netlink sockets are used to transmit messages between the user space routing daemon and the Linux kernel module. In the following, we discuss the technical details of netlink sockets and how the details influence the implementation of the kernel module.

Netlink sockets [He05] offer a datagram oriented service and enable user space processes to exchange information with kernel modules using the standard socket API [SFR03]. In kernel space, a lower level API is provided. Conceptually, netlink messages are sent and received asynchronously; packets are placed in a queue to smooth the burst of messages. However, when messages from user space to kernel space are sent using the `sendmsg` system call, the reception code in the kernel

module is invoked in the context of the sender as system calls require synchronous processing [He05].

This means that if we wish to have message processed asynchronously in the kernel module, the code must explicitly implement the processing in a kernel thread. The message is then served in another context, the `sendmsg` system call returns immediately, and other system calls can enter the kernel and thus, context switch granularity is improved [He05]. If asynchronous processing is not implemented, the netlink reception code should preferably return fast, as no other system call can enter the kernel while the reception code is running.

In the DYMO-AU kernel module, no kernel threads is created to serve the reception of netlink traffic in another context. Reception occurs in the context of `sendmsg`, which is called by the routing daemon. This design has been chosen to keep the code simple, and to avoid introducing subtle thread related bugs. However, no assumptions about message being processed synchronously have been made. It is possible that the implementation is changed to use a dedicated kernel thread for reception in future versions.

6.2.4 Netfilter Hooks

The netfilter hooks component is responsible for generating the events that trigger routing protocol action. This has been implemented as a function that is registered three different places within the netfilter framework.

Before we explain the work of the callback function at the three hooks, we note that in order for the implementation to function properly, it must not interfere with DYMO packets. These must be directly accepted without being subject to further processing in the netfilter hook function. To identify DYMO packets we consider the IP header of each processed packet and if the protocol is UDP, it is checked if the port number matches the port number used by DYMO-AU.

In section 4.2.3, we presented the appropriate netfilter hooks an implementation can register callback functions at, to be able to recognize the needed events identified in section 4.1. The DYMO-AU implementation follows the direction given in that section. That is, in our implementation we register a callback function at the NF_IP_LOCAL_OUT hook, the NF_IP_PRE_ROUTING hook, and the NF_IP_POST_ROUTING hook. This means that the callback function is called whenever a message is sent from a local process, a message is received on a network interface, or a message is sent on a network interface.

Figure 6.7 illustrates how we check if route discovery should be initiated. As mentioned in section 5.2.4, at the NF_IP_LOCAL_OUT hook, the function checks the destination address of packets sent from a local process against the expiry list and possibly tells the user space routing daemon to initiate route discovery. The function returns NF_QUEUE while route discovery is ongoing to ensure packets for the destination are buffered. Note that the routing daemon is only told to start route discovery if it has not done so before; this is the case if the packet queue is empty.

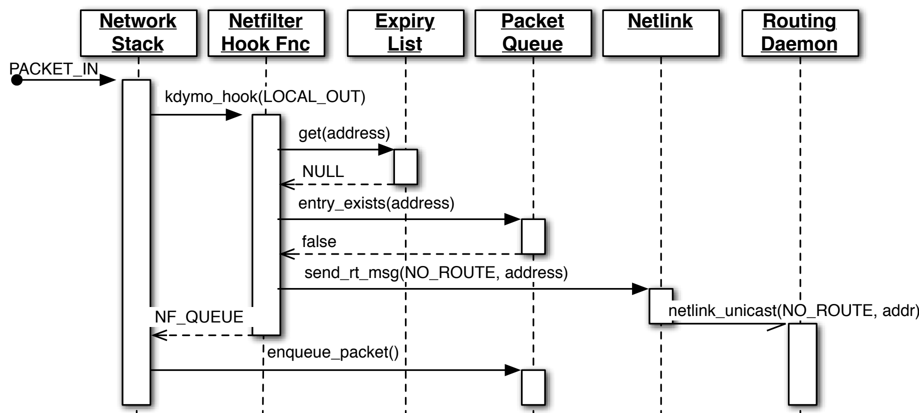


Figure 6.7: Checking for active routes in the kernel module when a local program sends packets. No route for *address* exists and the routing daemon is informed such that route discovery can be initiated.

At the NF_IP_PRE_ROUTING hook, we check that a route to the IP destination exists. If no route exists, an RERR message is generated. Furthermore, at the hook, we update the timeout for IP source address if a route for it exists. In figure 6.8, it is illustrated how the check for a route at the NF_IP_PRE_ROUTING hook is performed in the kernel module. In this particular example, there is no route for the destination, i.e., the expiry list returns NULL when queried for an entry. The kernel module netlink component is then told to send a message to the routing daemon, and a NO_ROUTE message is then sent to user space. We have not illustrated the update of the timeout for the IP source address.

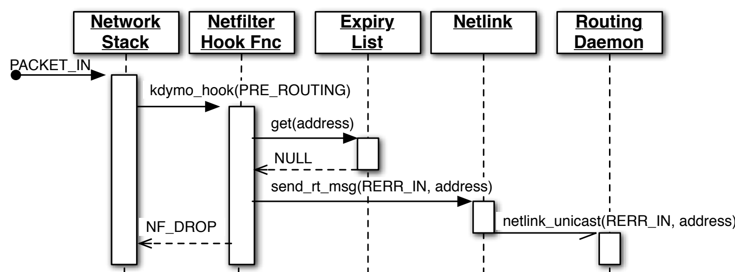


Figure 6.8: Checking for active routes when receiving packets on a network interface. No route for *address* exists and the routing daemon is informed. The check for an active route is similar to the check in figure 6.7.

At the NF_IP_POST_ROUTING hook, timestamps of active routes are updated when sending or forwarding packets. In this way, both locally generated packets as well as forwarded packets can be considered.

6.3 Updating Route Timeouts

In the following, we present three different ways of updating route timeouts and deleting stale routes. In all three cases, work takes place both in user space and in kernel space. However, the common distinctive feature is how fresh the valid timeout value of a route table entry in the routing daemon is, compared to the actual freshness information recorded by the kernel module. The greater the number of times the user space value is updated to reflect the value recorded by the kernel, the greater is the amount of data that must be exchanged between the routing daemon and the kernel module. For each of the methods we evaluate the benefits and drawbacks.

6.3.1 Packet-triggered Update of Timeouts

This approach ensures that all entries in the routing table of the routing daemon are updated with the latest registered value.

The procedure is simple. An update message is sent from the kernel module to the user space routing daemon every time a packet is received from a node or sent to a node with an active route. When the routing daemon receives the message, the valid timeout of the corresponding entry is updated and so is the valid timeout of the next hop entry.

Some time prior to the update of the valid timeout, a timer task that monitors the valid timeout of the route table entry has been scheduled (see section 6.1 for further details). Now we have two implementation options. The first is to take no further action. The second is to immediately cancel the scheduled timer task that monitors the valid timeout and subsequently schedule a new timer task based on the updated valid timeout value.

Using the first option, the daemon takes no further action. What then occurs is that when the task that checks the valid timeout eventually is executed, it detects that the timeout of the entry is valid, i.e., it is after the current time, and schedules a new timer task checking the valid timeout. The task is scheduled to execute when the valid timeout expires. This can continue as long as data is transmitted using the route. In figure 6.9, this case is illustrated. Note, however, that the periodical execution of a timer task that monitors the valid timeout is not shown nor is the update of the next hop entry.

Using the second option, the daemon does one additional operation after updating the valid timeout of the route table entry. Instead of waiting for the execution of the timer task that monitors the valid timeout, the timer task is cancelled immediately and a new task is scheduled using the newly updated valid timeout value. This case is illustrated in figure 6.10. Only the extra operations performed when calling updateValidTimeout is shown; everything that happens until then is similar to the operations shown in figure 6.9. After the valid timeout value of the entry is updated, the monitoring timer task is cancelled and a new one created and scheduled. Once again, the update of the next hop is not shown in the figure.

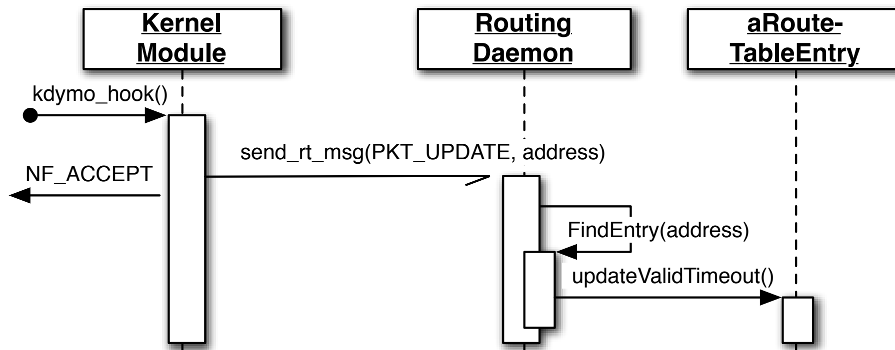


Figure 6.9: Updating the route timeout in the routing daemon when processing a packet in the kernel module.

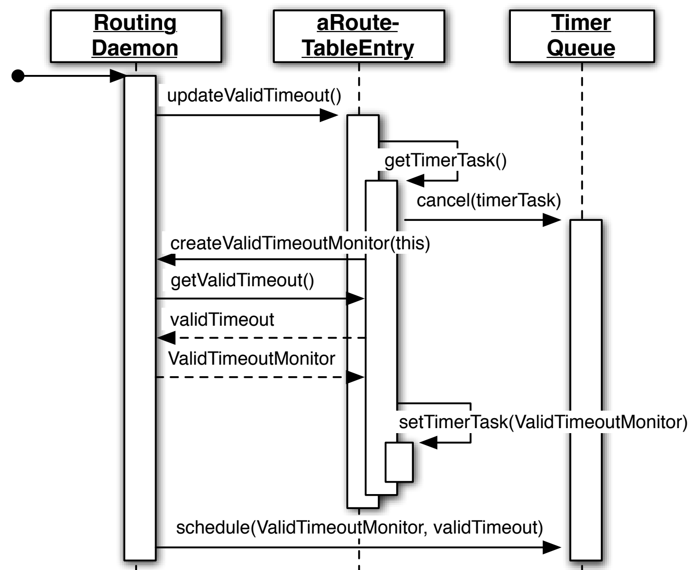


Figure 6.10: Updating the route timeout and rescheduling the valid timeout monitor timer task in the routing daemon when processing a packet in the kernel module

The idea behind this additional operation is that the routing daemon will impose additional load on the system if the timer task is not updated: This happens as the `select` function used in the main loop returns more frequently because the monitoring timer task is scheduled based on a prior timeout value, rather than the most recent value received from the kernel module. I.e., the idea is that while the route is active, the validity of the route should not be rechecked.

Both implementations have been used during testing and experimentation. The implementation that updates the timer task and timer queue at every timeout update was originally the solution implemented. However, we discovered that the idea of a possible advantage of trading-off fewer calls to `select` in the main loop in return of requiring cancelling and scheduling of a new timer task at every update was flawed. During experimentation, we observed that CPU usage, using the implementation that updates the timer task and timer queue at every timeout update, would peak at 90 percent. Without the constant update of timer tasks, the usage would peak at 2 percent. The reason for this extremely high CPU usage has not been investigated thoroughly, but we suspect that it is caused by how timer tasks are implemented. Whenever a timer task object is created, eight bytes of memory are dynamically allocated using `malloc`. The small size of the allocated objects can cause the implementation to spend a large proportion of its time in `malloc` and `free`.[1] However, we have not collected any hard evidence supporting this hypothesis.

We now leave the discussion of the implementation details of the *packet-triggered update of timeouts* method and continue with a discussion of the benefits and drawbacks of the method. The major benefit of *packet-triggered update of timeouts* is that one can always be sure that the timeout valid obtained by querying the routing daemon routing table is up-to-date. This is particularly valuable during initial testing and debugging and later while doing practical experiments. Second, it has a simple implementation; communication is one-way, message are sent from the kernel and received by the routing daemon. When a message is received by the routing daemon, no further processing are required by the kernel.

The greatest drawback is the overhead associated with transferring a message from the kernel module to the routing daemon whenever sending, receiving, or forwarding a packet. Forwarding even implies two update messages, as route timeouts are updated for both the sender and the destination of packet, which are implemented at two different netfilter hooks (see section 6.2.4).

Because of the above-mentioned advantages, the *packet-triggered update of timeouts* method is used in the current DYMO-AU implementation.

6.3.2 Timeout-triggered Update of Timeouts

This method only updates the valid timeout values for a route table entry in the user space daemon whenever the valid timeout monitor for an entry fires.

[1]Doug Lea [Lea00] describes an implementation of `malloc` that has this very property. A modified version of this implementation is a part of glibc, the Linux C library [Glo06].

Instead of checking the valid timeout value when executing as described in the preceding section and in section 6.1.8, the timer task sends a message to the kernel module asking for the current value as registered by the module. Upon receiving an answer from the kernel, either a new valid timeout monitor or a delete timeout monitor is scheduled. Hence, this method is similar to the one described previously with regards to scheduling of timer task. The major difference is that no valid timeout messages are transferred from the kernel module to the routing daemon when receiving or sending packets. The update procedure is illustrated in figure 6.11. After the timeout value has been received from the kernel, a new valid timeout monitor is scheduled.

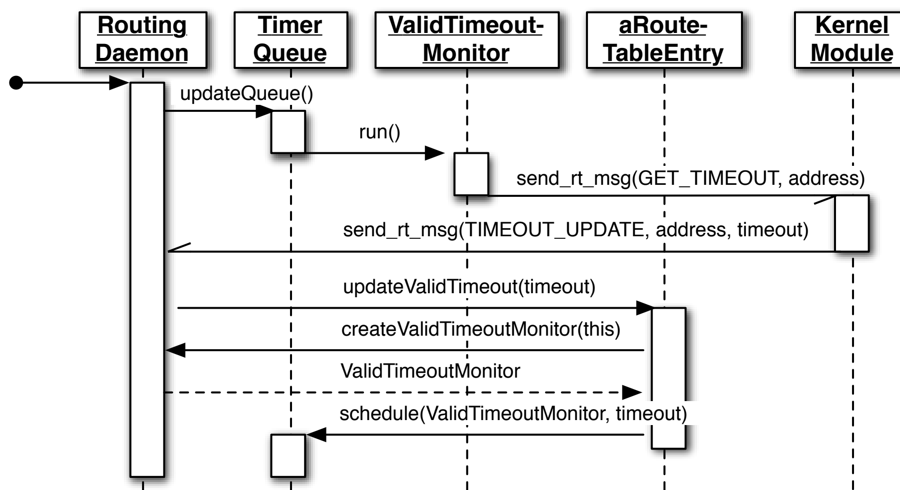


Figure 6.11: Updating the valid timeout value for a route table entry periodically polling the kernel module.

We must consider some obstacles for this scheme to work as intended. First, because of how network communication is handled by the daemon in which network reads are processed in context of the `select` call in the main loop, the reply messages from the kernel is processed asynchronously. That is, after a request is sent from the daemon, the code returns to the main loop and `select` call. Not until then can the reply message be read from the netlink socket. This is primarily a problem if a DYMO routing message is received simultaneously with the kernel module message and the subsequent processing of it requires an up-to-date value of the valid timeout for a route table entry. This is, for example, the case when the sequence number for a node found in the routing message is equal to the sequence number found in the route table entry for the node [CP06b].

A second problem is that the netlink protocol is not reliable. Messages may be dropped if out of memory conditions or other errors occur [Net99]. This can happen if either the kernel module or the routing daemon is not able to keep up with the stream of messages, i.e., the socket receive-buffer runs full. If either the

request or the reply message is discarded, the route table entry may never be deleted as a delete timeout timer task is never created and scheduled. One way to alleviate this problem is to request that the receiver acknowledged the messages. The netlink socket API provides means to facilitate this.

Two additional solutions exist besides the one sketched at the start of this section, i.e., that a timer task retrieves the current valid timeout of a route from the kernel module with the use of netlink messages. The first solution is to use the */proc file system* to get timeout values from the kernel. As mentioned in section 6.2, the /proc file system is a virtual file system that is used as an interface to the kernel. The kernel module exposes the contents of the expiry list through a file in the /proc system. By opening and reading this file from the routing daemon, it is possible to obtain the required valid timeout parameters synchronously. Using this solution avoids the two problems associated with netlink communication. This method was used and described by Kawadia et al. [KZG03].

The second solution is mostly designed to cater for the possibility of dropped messages. The idea is to create and schedule a timer task that monitors the delete timeout for a route table entry. The timer task is created at the same time as the route table entry. That is, two timer tasks are created simultaneously when creating a route table entry. One task monitoring the valid timeout value and one task monitoring the delete timeout value. If either a request or a reply message is dropped and no delete timeout is scheduled, the original scheduled timer will eventually delete the entry. This solution is still subject to problems because of the reply being received asynchronously.

The major advantage of periodically updating timeout values in the routing daemon is that far fewer messages have to travel from kernel space to user space. A disadvantage is the possibly added complexity involved in ensuring that the daemon is not left in an inconsistent state because of dropped control message. Obtaining state information from the kernel module reading a file from the /proc file system shares the same idea and advantage, but avoids the disadvantage as its implementation is simpler and no responses have to be processed asynchronously. Generally, the *timeout-triggered update of timeouts* method has the drawback that the route timeouts recorded by the daemon is not always up-to-date. The values can, however, be obtained from the kernel module through a file in the /proc file system.

6.3.3 On-demand Update of Timeouts

As described in section 6.2.2, the queue of expiry entries in the kernel is only used for look-ups. The data structure does not itself test if an entry in its queue has expired. It assumes that if an entry exists, then the route has a valid timeout. It is the responsibility of the user space daemon to tell the kernel to delete entries in the queue.

The on-demand update of timeouts method instead assigns responsibility of deleting entries in the queue to the kernel module. Every time the queue of expiry

entries looks up a node, it checks if the entry for the node has expired. If the entry has expired, it is deleted. As the routing daemon is not responsible for deleting expiry list entries anymore, it does generally not need to have an updated value for valid timeouts (except in the case mentioned in the previous section). In this case the current value can be obtained through the /proc file system as described in section 6.3.2.

However, one problem remains. An entry that is never used subsequently to its addition in both the routing daemon and kernel module, will never be deleted from either places. To cover this case one could schedule a delete timeout monitor timer task as proposed in section 6.3.2. This timer task would periodically (for instance, every 30 or 60 seconds) check the validity of the kernel module entries and make efforts to delete any entries with valid timeout before the current time plus DELETE_ROUTE_TIMEOUT.

The difference compared to the method previously described in section 6.3.2, is that only *one* timer task is required to flush stale entries. The kernel module would delete any stale routes on demand, if a packet was attempted transmitted for a destination with an invalid route.

To summarize: expiry list entries are deleted directly by the kernel module if they have expired when they are trying to be retrieved. This is unlike the methods described in sections 6.3.1 and 6.3.2 in which entries were deleted on request from the routing daemon. To avoid spurious entries and memory leaks, entries will periodically be checked and deleted by the routing daemon.

The advantage of only updating route timeouts on-demand, is that hardly any messages are transmitted between the routing daemon and the kernel module that would not have been sent otherwise, i.e., when route discovery must be initiated or when RERR messages must be broadcasted. The disadvantage is that the route timeouts recorded by the daemon is not up-to-date.

7 Experimental Evaluation

Compared to simulation studies, relatively few measurement experimental studies have been performed with mobile ad hoc networks deployed on real hardware. Evaluating applications and communication for ad hoc wireless networks typically involves simulation. When experimental evaluations are performed, it often means small-scale live deployments. Larger-scale evaluations have been performed, but it is typically costly and difficult to conduct under controlled conditions. Small-scale evaluations are easier to perform and our experiments are an example of a small-scale evaluation. In this chapter, we present the experiments conducted to perform practical evaluation of our DYMO-AU implementation and the DYMO ad hoc routing protocol.

The outline for the rest of the chapter is as follows. In section 7.1, we survey earlier practical evaluation of on-demand routing protocols and routing protocol implementations. The examined experiments are used as a basis for our own experiments. We also survey a representative set of proposed MANET software and hardware testbeds to explore the various solutions available for practical experiments with MANET implementations. We describe the chosen set of experiments and the experimental setup in section 7.2. In the following sections, we present the results of the experiments. In section 7.3, we present the results for route discovery latency, in section 7.4, we present the results for UDP performance, and in section 7.5, we describe measurements of the end-to-end delay or round-trip time. In section 7.6, we present the results on TCP performance and we conclude the presentation of the experiments in section 7.7, by describing some special TCP experiments that aim at measuring the so-called ad hoc horizon in a set of topologies called beam star topologies. Finally, in section 7.8, we present some of the experiences obtained while conducting the experiments. It should be noted that the word testbed can be used to denote two different things. First, it can be used to both denote software, hardware, or machinery used to facilitate and ease experiments and second, it can be used to denote the actual physical location of where an experiment is carried out.

7.1 Related Work and Testbeds

Some of the first experimental evaluations of ad hoc networks were carried out by Maltz et al. [MBJ99, MBJ00]. They conducted the experiments on a 700 m by 300 m outdoor testbed consisting of two stationary nodes placed at each end approximately distanced at 700 m and five mobiles nodes driving back and forth between the two stationary nodes. The wireless hardware used for the experiments are pre-802.11 WaveLAN radios. The implementation used was an implementation of DSR running as a part of a modified FreeBSD kernel (see also section 4.2.1). Part of Maltz et al.'s experiences of using the testbed was related to debugging and testing their implementation, but they also did evaluations on radio propagation and verification of some of the algorithms used by DSR.

With regards to performance evaluations, Maltz et al. performed simple experiments with ping and measured the packet loss ratio. TCP experiments with single-hop and two-hop topologies were conducted and the average throughput was measured with 1 MB and 5 MB file transfers. The experiments were conducted both in an indoor lab setting and on the outdoor testbed. In the two-hop experiments, measured throughput was 25 % of the indoor lab measured throughput. Indoors, the nodes are in the same collision domain, so there are no hidden terminal problems whereas in the outdoors setup, the hidden terminal problem was introduced.

Lundgren et al. conducted some large-scale experiments with up to 37 nodes in both indoor and outdoor environments [LLN$^+$01, NGL05]. Their primary goal is to characterize mobility scenarios with a proposed metric for mobility, the *virtual Mobility* (vM) metric, to be able to compare and distinguish between the scenarios. The vM metric is calculated based on measured signal quality between nodes. In addition to measurements of signal quality, some experiments were also made to assess the ability of two early implementations of AODV and OLSR to establish multi-hop routes by measuring packet loss and analysing the number of hops reached when using the ping utility. Similar to the experiments conducted by Maltz et al., the experiments carried out by Lundgren et al. were made with pre-IEEE 802.11 WaveLAN hardware.

Lundgren et al. built the *Ad hoc Protocol Evaluation* (APE) testbed in order to be able to perform the large-scale experiments in a reproducible fashion. APE offers a platform based on the Linux operating system specially tailored for ad hoc routing protocol evaluations with the intent to allow for greater repeatability and reproducibility. APE uses a scenario interpreter that executes commands, including data traffic generation, at specified points in time. It introduces choreography scripts providing instructions to testers who walk around with mobile nodes. The intent is to test routing protocols and various topologies in real physical setup as compared to an emulated setup using, for instance, packet filtering. While experiments are carried out, extensive logging takes place on each node. When finished, all log files are collected and merged at a central node, and APE then provides tools to process and extract information from the log files.

Gray et al. [GKN+04] did several outdoor and indoor experiments with up to 33 nodes with 4 different implementations of MANET routing protocols, among them AODV and compared the results to simulations. The experiments evaluate four different metrics: message delivery ratio, control packet overhead, route efficiency, and end-to-end delay. Comparing the results obtained outdoors with the results obtained indoors, AODV showed much improvement with regards to packet delivery rate when going indoors, while ODMRP, a multicast protocol that uses broadcasting showed much worse packet delivery ratio. Indoor, the mobile nodes were placed in such a way that each node could hear all transmitted packets. GPS information recorded in the outdoor settings was used to simulate network connectivity. Indoor, ODMRP was affected because of an increased number of packet collisions as the IEEE 802.11 RTS/CTS clearing procedure is not used for broadcast packets [Gas02].

Desilva and Das [DD00] implemented the AODV protocol as an extension to the ARP protocol and measured the performance of their implementation using UDP and TCP traffic. They tested a static setup consisting of one desktop machine and four notebooks arranged in a chain topology. Desilva and Das measured route discovery latency and received throughput of UDP and TCP traffic. In the route discovery latency experiments, in an unloaded network, a neighbour could be discovered in 3–4 ms and each additional hop would add roughly 4 ms to the delay. In a network loaded with UDP traffic, route discovery latency was much higher and discovering a node four hops away would on average take 700 ms. When using UDP, the offered load vs. measured throughput were recorded. The experiments were repeated with different values of offered load and with various number of hops. For each node in the chain, the number of packets dropped by the node and between its predecessor and successor was recorded to assess the effect of hidden terminal problems. To measure TCP performance, a 14 MB file was transferred with FTP from one end of the chain to the other. The node farthest away was at a distance between one and four hops. Desilva and Das found that they had to carefully place the nodes in order not to experience severe packet loss and stalled TCP sessions. In their experiments, as the hop count increased the transfer got progressively slower. As the hop count increased, more and more packets were dropped leading to retransmission and duplicated ACKs.

Kuladinithi et al. [KUFG] also used a setup with stationary nodes to compare two implementations of the AODV routing protocol as well as an experiment with static routes in which route table entries are added manually. A six-node testbed was used with a chain topology varying the number of hops from one to five. Experiments using both UDP and TCP were performed. In the UDP experiment, numbers for offered load vs. received throughput, packet delivery ratio, jitter, and packets arriving out of sequence were recorded. In the TCP experiment, throughput was measured. In addition, a special patched version of the Linux kernel was used to measure deviation in the retransmission timeout (RTO) value and the size of the congestion window (CWD). The values of the RTO and the CWD over time were used for investigating a large number of observed duplicate TCP ACKs. One in-

teresting observation made was that TCP throughput was better when testing with one of the tested AODV implementation compared to statically assigned routes.

Gupta et al. [GWW04] studied the practical performance of TCP in a multi-hop wireless ad hoc network environment using a signal-strength aware implementation of AODV. Experiments were primarily conducted using five stationary nodes set up in a chain topology but using a varying number of hops. Experiments involving one moving node was also conducted. The different metrics measured were TCP throughput, route discovery latency, control packet overhead, and end-to-end delay. Generally, they found that increasing the number of hops, or size of packets affected performance. For TCP performance, they found that hidden node and exposed node problems influenced 3-hop and 4-hop configurations. Their observed route discovery times ranged between 51 ms and 613 ms which are considerably higher when compared to Desilva and Das [DD00]. The End-to-end delay showed increased round trip times with more hops as well as when increasing the packet size.

Bae et al. [BLG00] measured unicast UDP performance in a seven- node testbed setup using the *Ordered On-Demand Multicast Routing Protocol* (ODMRP). They used a network with stationary nodes with no mobility as well a setup with two kinds of mobility. Using the setup with stationary nodes they compared received throughput in experiments with statically assigned routes and experiments in which traffic was routed with the ODMRP protocol. In the latter case, control packet throughput and overhead was also measured. Similar experiments were performed with the mobility scenarios.

Tschudin and Osipov [TO04] investigated the usefulness of TCP based network services running on top of IEEE 802.11 networks in which routing is controlled by a MANET routing protocol. Usefulness is defined in terms of the *ad hoc horizon*, which is the number of hops and number of participating nodes beyond which performance severely degrades and the service stops being usable for an end user. The experiments defined by Tschudin and Osipov are performed in a set of topologies named *beam star* network topologies. A central node acts as a server and this server is the origin of paths or beams that end at nodes distanced an identical number of hops away. Two types of experiments were performed. In the first experiment, a number of parallel FTP sessions are present. The central node starts FTP transfer from each end-node. The various sessions then compete for the bandwidth. In the second experiment, one FTP transfer is started and the other end-nodes request a 36K web page from the central node every 15 seconds. Tschudin and Osipov use the ns-2 simulator for the experiments and find the above-mentioned horizon to be 2 to 3 hops and approximately 15 nodes.

7.1.1 Evaluation Testbeds

Before beginning our experimental tests and evaluations we have examined the various testbeds that have been proposed for MANET practical experimentation and evaluation. In what follows, we make an outline of a representative set of

proposed testbeds.

Testbeds Requiring Special Hardware Several testbeds requiring use of special hardware have been proposed [KR01, JS04]. The common goal is the desire to reduce the transmitting range of wireless radios to allow wireless nodes to be placed within much closer proximity and thereby eliminate the requirement of a large area for conducting experiments.

Kaba and Raichle's "testbed on a desktop" [KR01] uses the ability to connect external antennas to IEEE 802.11b wireless PC cards as a way to control radio propagation. The communication range was reduced with the external antenna instead of extending it. The signal was attenuated and low gain antennas were connected to the wireless card hereby disconnecting the internal antenna. The wireless card was furthermore shielded as a safety measure to prevent signal leakage from the internal antenna. Kaba and Raichle experienced various radio propagation effects while conducting experiments. People or objects moving around influenced network topology making it difficult to control the experiments. To be able to control this effect, a more advanced wired testbed was devised, which is to be created with the use of splitter/combiners, additional attenuators, terminators, and cables, wiring the wireless card antennas together. As an advanced configuration, a control computer could control the attenuation levels.

Judd and Steenkiste [JS04] employs a similar approach as the above described devised wired testbed by Kaba and Raichle, but relies on digital emulation signal propagation using a *field-programmable gate array* (FPGA) *Digital Signal Processing* (DSP) engine, local oscillators, and D/A and A/D boards. Signals coming from wireless nodes are mixed, digitized, and fed to the FPGA-based DSP engine. In the DSP engine, the signals are processed to emulate a physical environment. Signals are then reconstructed, mixed, and finally sent to the wireless card antenna port of the destination(s). Their prototype furthermore uses an emulation control computer that based on a given scenario computes attenuation values for the wireless nodes which is then fed into the FPGA. The emulation controller is also used to allow remote management of the wireless nodes using a separate control network. This way, the controller can coordinate the traffic generated by the nodes.

In its basic setup, Kaba and Raichle's approach has the advantage that it is possible to move nodes around. However Judd and Steenkiste goes beyond their more advanced devised setup as they have incorporated an emulation controller in their setup that allows them to dynamically set attenuation levels and control traffic generation on the wireless nodes.

The ATMA framework proposed by Ramachandran et al. [RABR05] seeks to assist with the management of nodes in a wireless testbed by deploying a multi-hop mesh network, which is used to monitor testbed nodes. The primary motivation for using a mesh network alongside the testbed is to allow management traffic to be transmitted out-of-band rather than in-band, i.e., where traffic is a portion of the overall testbed traffic. For each node in the testbed, one additional management

node is required. This only makes ATMA feasible when deploying a dedicated testbed in places with absolutely no existing infrastructure, as the hardware cost would be prohibitive if only conducting occasional experiments. With regards to pre-existing infrastructure, if, for instance, Ethernet is available, out-of-band traffic can be transmitted on a wired interface instead. Some management tools are provided to control the testbed nodes among them a tool to filter packets.

Software Testbeds Zhang and Li developed MobiEmu [ZL02], an environment that emulates connectivity in a wireless network by filtering packets before they traverse the operating system network stack and thus, before reaching the routing module. The emulator is scenario driven accepting mobility scenario files generated with CMU's *setdest* tool, which is included in the ns-2 distribution. A preview of the events recorded in the scenario file can be visualized in a GUI front-end before using the scenario on the testbed nodes. MobiEmu uses a trivial propagation model where any nodes falling within a 250 meter radio range of each other can communicate, and any falling outside the range cannot. MobiEmu does not emulate the physical and MAC layer, which makes it less suitable for performance evaluations. Its stated goal is to aid in while developing and debugging MANET routing protocol implementations.

For its functioning, MobiEmu employs a master controller component that controls the time in the network. The current time is multicasted whenever the controller detects a link break or link formation. When the slave controller component running on the testbed nodes receives the time stamp, it updates the packet filters based on the topology rules, by either adding or removing connectivity to a node. Master/slave management communication should be transmitted on a control channel separate from the testbed network to avoid having control traffic in-band. The wish to separate control traffic from data traffic is similar to that of ATMA. Furthermore, the master controller node can only run the slave controller component concurrently if a separate control network is used. MobiEmu does not support traffic generation on testbed nodes, however, with some work and with some restrictions, traffic generation can be incorporated using home-made utilities.

The *MANET Testbed Manager* (MTM) tool proposed by Lent [Len05] is similar to MobiEmu, but extends its modes of operation as it also wishes to emulate the physical layer. One goal is to be able to use the tool for realistic performance evaluations. MTM supports more advanced scenario definition in which obstacles that obstruct both physical movement and wireless communication can be included. MTM employs ray tracing support and obstacle modelling to model radio propagation. Also included is a ground reflection model that trades-off accuracy for execution speed, but also allows comparisons with experiments conducted with ns-2. Lent lists diffraction, scattering, and small-scale fading effects on radio wave propagation as future work. In addition, simulations of hidden and exposed terminal problems, the effects of employing nodes with different transmission power, and the use of different power levels based on whether transmitting using unicast

or broadcast are also listed as future work. The MTM tool looks promising as it enables the use of an emulator tool for performance measurements. At time of writing, a running version of the proposed MTM tool has not been released.

7.1.2 Summary

In this section, we have given an overview of previously performed practical experimental evaluations. An outline of a representative set of testbeds has also been given. To summarize, we have listed the described experiments and the associated measured metrics in table 7.1. A dash (–) in the third column of the table means that no special software was used or that the authors did not specify the software used.

Experiment by	Measured metrics	Software used
Maltz et al.	UDP: Jitter, Message delivery ratio, TCP throughput	macfilter (packet filter)
Lundgren et al.	virtual Mobility, Packet loss and multi-hop setup	APE testbed
Gray et al.	Message delivery ratio, Control packet overhead, Route efficiency, End-to-end delay	–
Desilva and Das	Route discovery latency, UDP and TCP throughput	–
Kuladinithi et al.	UDP: throughput, message delivery ratio, jitter, out-of-order packets, TCP throughput,	–
Gupta et al.	Route discovery latency, Control packet overhead, End-to-end delay, TCP throughput	–
Bae et al.	UDP throughput, Message and control packet delivery ratio, Control packet overhead	–
Tschudin and Osipov	Unfairness ratio, Download time	ns-2 (simulation)

Table 7.1: The experiments described in this section.

7.2 Experiments

The experiments that we perform and the chosen test topologies are decided, based on what has been done in prior experiments and experimental evaluations as outlined in the previous section. Practical limitations also play a role: what scenarios

are practical when the tests are to be carried out by only one person. The set of chosen experiments are:

- Route Discovery Latency
- UDP Throughput
- End-to-End Delay
- TCP Throughput
- Unfairness Ratio and Download Time (Beam star setup)

With regards to the chosen network topology, we have primarily used a testbed setup with no mobility in which laptops placed in a chain where each laptop is only able to communicate with its immediate neighbours. This setup was chosen as it had previously been used in experiments conducted by Kuladinithi et al., Desilva and Das, and Gupta et al. Furthermore, as we had only access to a limited number of laptops, using this setup allowed us to experiment with the maximum possible number of hops. A chain topology with six nodes is depicted in figure 7.1.

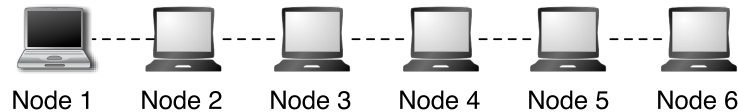


Figure 7.1: A five-hop chain topology.

To provide an additional basis of comparison we find it interesting to investigate how the proposed experiments perform using one of the evaluation testbed described in section 7.1. With regards to selecting on the proposed testbeds, all testbeds that rely on special hardware can be rejected as they generally require custom tailored hardware and are not reusable outside the specific environments. The options remaining are the software-based testbeds. The APE testbed is primarily useful for large scale testing and the use of choreography scripts also requires a lot of manpower. The only serious available testbed is *MobiEmu* by Zhang and Li [ZL02]. Thus, MobiEmu has been used in our practical experiments as well as the final testing and debugging process of our DYMO implementation.

Most of the experiments have been performed in two different setups. In the first setup, the laptops are placed in hallways, properly physically distanced so only the immediate neighbours of a node are able to transmit data to the node. This setup is called the *real setup*. In the second setup, the laptops are all placed in the same room, and the packet filtering capability of MobiEmu is used to obtain the same effect as if the laptops were placed physically separated. We call this setup the *MobiEmu setup*. Using MobiEmu in this fashion, for example, when a node broadcast a DYMO RREQ message, the packet filter ensures that only the

neighbours are able to process the packet. At all the other nodes, the packet will be dropped before it traverses the network stack.

The reason MobiEmu has not been used exclusively for the practical evaluation experiments is that it cannot be used to obtain realistic numbers. When placed within the same room, the wireless radios of the nodes are within the same collision domain and using link sensing, it determines when the other nodes send data and consequently, no nodes are sending unicast traffic simultaneously. We are experiencing no hidden terminal and exposed node problems.

7.2.1 Experimental Set Up

We briefly describe the hardware and software used for the experiments.

Hardware The testbed consisted of six computers. Of these, five were IBM Intel Centrino ThinkPad laptops with a 1.5 GHz Pentium M processor and 512 MB of RAM. Each ThinkPad had an internal Intel PRO/Wireless 2100 Network Connection mini PCI adapters for IEEE 802.11b wireless connectivity. The wireless card uses the *ipw2100* driver on Linux [Ipw06]. The last laptop was a Dell Inspiron with a Pentium III 1.2 GHz processor and 512 MB of RAM. The laptop has a built-in wireless network adapter using the Intersil Prism II chipset and using the *orinoco* driver [Tou04].

In the experiments the wireless cards on the laptops was set to channel 9 to avoid interference with the standard access points channels 1, 6, and 11. The transmit rate for the cards was configured for automatic rate selection. Unintentionally, the *Request to Send* (RTS) threshold setting was left in its default *off* configuration which means that 802.11 RTS/CTS clearing procedure described in the beginning of chapter 2, was not enabled during our experiments. Only after the experiments had been conducted was it discovered that the RTS/CTS clearing procedure had not been enabled.

The experiments were initially performed using all six laptops. However, it quickly became clear that the throughput achieved when using all six laptops was considerable lower compared to experiments where only the five IBM laptops were used. As the Dell laptop was identified as the bottleneck, it was decided to also perform the experiments with only the five IBM ThinkPads to explore the maximum obtainable throughput when using the same type of hardware. In the following, when we present our result, in each specific example, we make clear whether all six laptops or only the five IBM ThinkPads have been used.

Software On all laptops, the Ubuntu 5.10 Linux distribution was installed (Linux kernel version 2.6.12). All nodes were configured with addresses in the 192.-168.42.X sub-network. The Dell laptop was assigned the 192.168.42.50 address and the IBM laptops were assigned addresses in the 192.168.42.51–55 range. When using the Dell laptop, it was either origin or target of data.

The *Iperf* network traffic generator [TQD+05] was used to generate UDP and TCP traffic. A network traffic generator is a device, or program used to measure various network properties, such as amount of available bandwidth, latency, and network jitter. Other network traffic generators considered were *D-ITG* [ITG] and *netperf* [Jon]. D-ITG has a number of sophisticated parameters to control the generated traffic that would make it an obvious choice, but the flexibility makes it cumbersome to generate traffic at a specific rate.

Gupta et al. [GWW04] also used netperf. The main feature of netperf is the ability to measure networking performance, for example, by measuring bulk data transfer performance, i.e., how fast can one system send data to another and how fast can the other system receive it. However, it is not possible to request a specific bit rate using netperf. Because of the inability or difficulty of specifying a bit rate, both D-ITG and netperf were rejected in favour of Iperf.

Test parameters The DYMO routing protocol parameters used in the test were all as defined in the DYMO specification revision 4 [CP06b]. One limitation in the DYMO-AU implementation that will limit the comparability of our tests with others, is the lack of a link sensing mechanism implementation, for example, Hello messages. This means when we have found a route and as long the route is still active, there is no routing protocol overhead on the link layer as no routing messages are sent periodically.

7.3 Route Discovery Latency

The definition of route discovery latency we use is defined as the elapsed time between the kernel module discovers that no route to the destination currently exists and when the kernel module is notified about the insertion of the route to the target destination into the routing table. This is not the same definition as used by Gupta et al. They use the following definition: “The route discovery time is the elapsed time between sending an RREQ and receiving the corresponding RREP.” [GWW04]. Desilva and Das merely note: “The route discovery latency is the time to discover routes” [DD00]. The difference between our definition and Gupta et al.’s definition depends on how long it takes for the processing that goes before sending the RREQ and after receiving the RREP. To be directly comparable to Gupta et al.’s results we would have to measure these processing times and subtract them from the obtained results. We have not measured these processing times, so our result are not directly comparable to Gupta et al.’s results, however, we expect the processing times to be within 1–10 ms range so these are negligible compared to measured route discovery latencies.

Before beginning route discovery, any cached link layer addresses are deleted from the ARP cache to ensure that ARP requests are being sent at every attempt. On intermediate nodes, the value of the Linux kernel parameter *gc_stale_time* has been set to 15 seconds. This means that after 15 seconds entries in the ARP cache

will change state to stale, which in return means that the link layer address will have to be verified [Bro03]. The interval between each route discovery attempt was 35–40 seconds.

7.3.1 MobiEmu Setup

In this section, we present the route discovery latencies for the MobiEmu setup. Route discovery latencies were measured in two different configurations. In the first, route discovery latency was measured in a setup with no traffic in the network. In the second, the route discovery latency was measured in a network with simultaneous UDP traffic.

The experiments with no load were conducted with both five and six nodes. We present the results for both setups separately as well as present a combined result in which the average and standard deviation have been calculated based on the measurements from both setups for a given X-hop experiment. For example, the result for the combined 4-hop experiment was calculated based on the union of the samples from the 4-hop experiment in the 5-node setup and the 4-hop experiment in the 6-node setup. For the 5-hop experiment, the combined result is identical to the 6-node result.

For each of the route lengths, at least 285 route discovery attempts were made, however, the 3-hop experiment with 6 laptops was only repeated 51 times and the 4-hop experiment with 5 laptops was only repeated 33 times. The results are shown in table 7.2.

Number of hops	Average Latency (ms)			Standard Deviation		
	5-node	6-node	Combined	5-node	6-node	Combined
1	49.3	137.3	87.1	130.8	82.6	120.7
2	112.1	736.8	313.8	72.4	800.9	543.6
3	214.9	492.6	255.4	80.4	196.1	143.7
4	361.8	404.9	400.6	96.7	188.3	181.6
5	–	823.7	823.7	–	230.1	230.1

Table 7.2: Average route discovery latencies for various numbers of hops (with no load).

The average route discovery latency numbers for the five-node setup shows that for each additional hop, the latency increases approximately 60–150 ms per hop. The numbers for the six-node setup are less clear-cut. For the 2-hop experiment the average latency is 736.8 ms, which is an increase of about 600 ms compared to the 1-hop experiment, but the average latency for the 3-hop experiment is only 492.6 ms. However, the standard deviation for the 2-hop experiment data is much larger compared to the other experiments denoting a lot of variability in the data, the maximum route discovery delay being 3350 ms (the highest delay for the other experiments is 1903 ms). When combining the two data sets from the two setups,

we see that the route discovery latency increases as the hop count increases, but the 2-hop result from the six node setup influences the combined result showing route discovery latency to be smaller over 3-hops than compared to 2-hops.

Table 7.3 shows the route discovery latency numbers obtained with simultaneously multi-hop UDP traffic. In this setup, the first node in the chain continuously sends UDP packets (1470 bytes) to the last node in the chain at a rate of 0.6 Mbit/s. Because of the UDP stream, a node will always know a route to its neighbour and consequently no numbers have been obtained for the 1-hop setup. Furthermore, as the first node sends UDP packets to the last node in the chain, no experiments have been performed for the 5-hop setup. For this reason we only have numbers for 2, 3, and 4 hops. The experiment was repeated between 33 and 105 times.

Number of hops	Average Latency (ms)	Standard Deviation
2	406.4	634.8
3	927.2	1548.6
4	1161.5	968.0

Table 7.3: Average route discovery latencies for various of hops (with multi-hop UDP).

Compared with the experiments without traffic (table 7.2), the latencies are higher, especially for the 3-hop and 4-hop experiments. The standard deviations are also higher, denoting a lot of variability in the data.

7.3.2 Real Setup

The numbers for the real setup were obtained while testing UDP performance, but the test conditions were similar to the ones used with the MobiEmu setup, i.e., the interval between route discovery attempts was 35–40 seconds to allow route table entries to expire from the DYMO daemon routing table. The *gc_stale_time* (see the beginning of this section) kernel parameter was set to 15 seconds to allow ARP cache entries to expire. The measurements were obtained when a node sends the first UDP packet, which mean the numbers were obtained with no load in the network. The results are shown in table 7.4 (unfortunately, no results were recorded for the 1-hop experiment with five nodes).

Taking into account the results obtained in the MobiEmu setup, the results presented in table 7.4 are as expected, except for the similar latency for the 3-hop and 4-hop experiments with six nodes. As the hop count increases, so does the route discovery latency.

If we compare the numbers to the one obtained by Desilva and Das [DD00], their measured route discovery latencies are much lower than ours are. For example, the measured average latency to discover a node 4 hops away is 14.14 ms. The implementation used by Desilva and Das is, however, different from ours as it uses the kernel modification approach and modifies the in-kernel ARP implementation

Number of hops	Average Latency (ms)			Standard Deviation		
	5-node	6-node	Combined	5-node	6-node	Combined
1	–	108.4	108.4	–	92.1	92.1
2	284.5	233.1	251.3	207.7	195.6	199.3
3	294.9	667.9	486.7	60.3	356.0	317.6
4	545.4	647.2	601.7	113.4	424.3	324.8
5	–	825.8	825.8	–	761.1	761.1

Table 7.4: Average route discovery latencies for various numbers of hops (with no load).

to provide support for AODV control packets. Gupta et al. [GWW04] report an average route discovery latency for a 4-hop setup of 613.3 ms using a modified version of the AODV-UU implementation [Nor]. This is almost identical to our results. The reported numbers for 2-hop and 3-hop experiments were approximately 50 ms and 300 ms, respectively. With regards to the 2-hop experiment, this is about 200 ms better than our result and regarding the 3-hop experiment it matches our experiment conducted in the 5-node setup, but is better than both our 6-node setup result and our combined result.

7.3.3 Comparing MobiEmu and Real Setup Results

To give a better overview of the results, the route discovery latency numbers for the MobiEmu setup and the real setup are depicted in figure 7.2. For the 1-hop experiments and especially the 5-hop experiments, the latencies are almost identical. The numbers for the 2-hop experiments are comparable. Looking at the results for 3-hop and 4-hop experiments, the MobiEmu setup has an advantage of about 200 ms. Now, if we assume that the result for the 2-hop experiment from the MobiEmu setup is exceptional in that, if the experiment was repeated it would yield a number lying between the 1-hop and 3-hop numbers, we would achieve a number below the 2-hop real setup. With this assumption, we conclude that route discovery latencies in the MobiEmu setup are better than in the real setup.

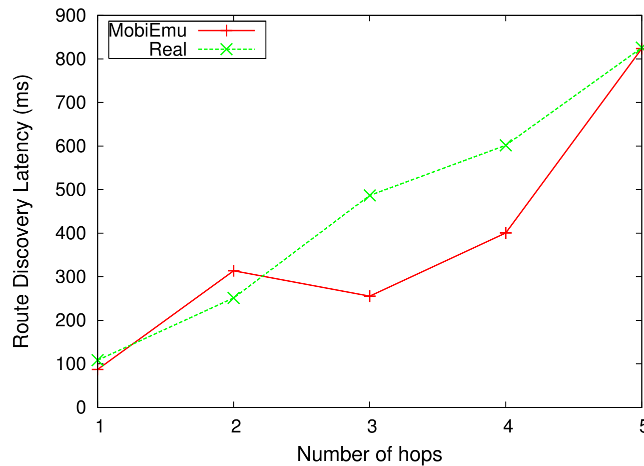


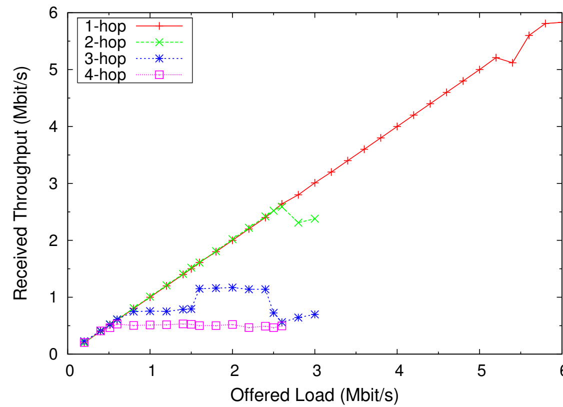
Figure 7.2: Average route discovery for the combined results for both MobiEmu and the real setup.

7.4 UDP Performance

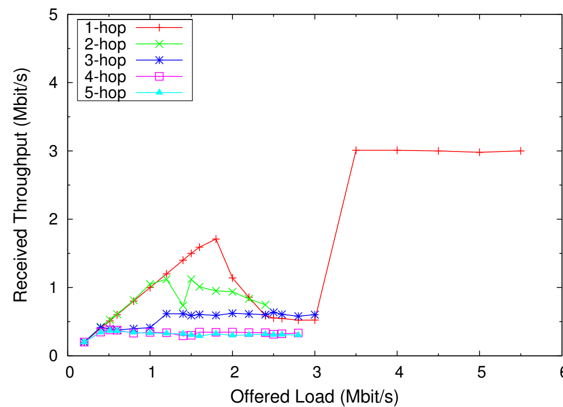
As previously described in section 7.2.1, the UDP performance experiments were conducted with both five and six nodes. In each test, the first node in the chain sends UDP packets at a constant rate to the last node in the chain. Which node is the last in the chain depends on the number of hops in the current experiment. For both the five-node setup and the six-node setup, the experiment was repeated with various data rates ranging from 0.2 Mbit/s to typically 3 Mbit/s with 0.2 Mbit intervals and additionally including the rates 0.5, 1.5, and 2.5 Mbit/s. In some experiments, the upper limit on the offered load was extended from 3 Mbit/s to 7 Mbit/s when received throughput exceeded 3 Mbit/s. For each of the data rates, the experiment ran for 60 seconds. The received throughput, a weighted jitter sample, and the number of packets received out of sequence were recorded for each second of the run. In the following, we only present the received throughput. The experiments were done in both the MobiEmu and the real setup. In the real setup, the experiments were conducted both with DYMO and with manually assigned routes in which the routing table entries are added before beginning the experiment. We also use the term static setup for this type of experiment or refer to the routes as being static because the routes are fixed during the experiment. In addition, we use the word static for the labels in the figures in this chapter.

Because of space considerations, we primarily present the experiments conducted with six nodes in this section. The trends observed in the five-node setup are similar to the six-node setup, but the numbers for received throughput for the five-node setup at identical hop counts, are higher. This can be seen comparing the two figures of figure 7.3, which shows the obtained received throughput for the real world five-node setup and six-node setup, respectively. In the 5-node setup, the

maximum received throughput of the 1-hop experiment peaks just below 6 Mbit/s while the corresponding 6-node experiment, the received throughput is a little more than 3 Mbit/s at its highest.



(a) Setup with five nodes.



(b) Setup with six nodes.

Figure 7.3: Offered load vs. Received throughput for the multi-hop UDP experiment with five and six nodes.

Looking at figure 7.3b we see that the received throughput for the 1-hop experiment drops quickly at around 2 Mbit/s offered load. The received throughput then stabilizes at around 0.5 Mbit/s received throughput until 3.5 Mbit/s offered load when the received throughput suddenly rises to 3 Mbit/s where it stays. It should be noted that the results for the offered loads higher than 3 Mbit/s are obtained at a different time in a separate experiment than the result for 3 Mbit/s and below. We do not have an explanation for this sudden drop in throughput from around 2 Mbit/s offered load. Compared to the 1-hop results in the 5-node setup depicted in figure 7.3a, we would expect the received throughput in the 6-node setup to continue to rise until stabilizing at around 3 Mbit/s or perhaps to stabilize just below 2 Mbit/s offered load where the drop occurs. Instead we observe the drop. Our

guess is that some external entity changed the radio propagation conditions. At some point, we observed that entering the office the sending laptop was placed in resulted in reduced throughput. However, we do not know if a similar incident takes effect in this specific experiment.

The results for received throughput from the above described experiments are shown in figure 7.4 and figure 7.5. The result from the 2, 3, 4, and 5-hop experiments are each shown in its own figure. The figures show the results for DYMO in the MobiEmu setup and real setup as well as the result for the manually assigned routes in the real setup.

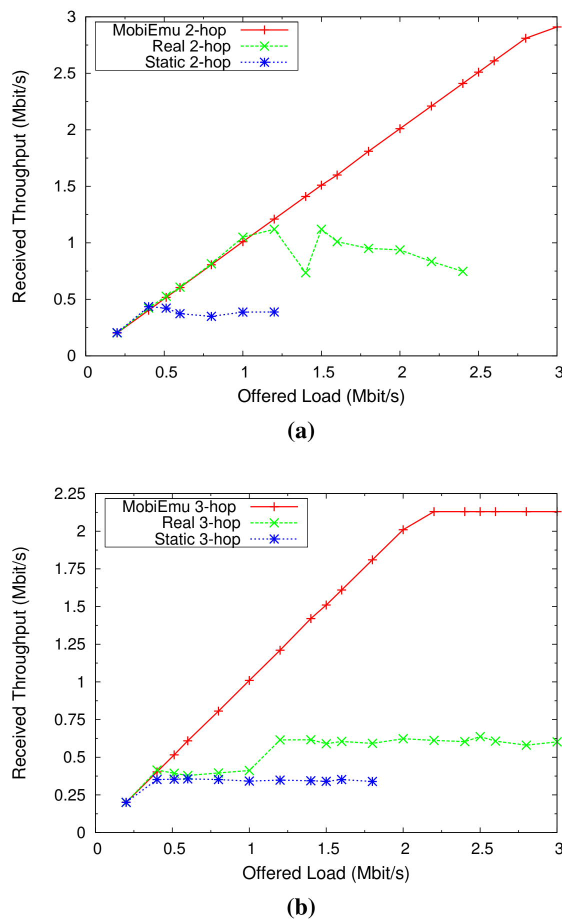
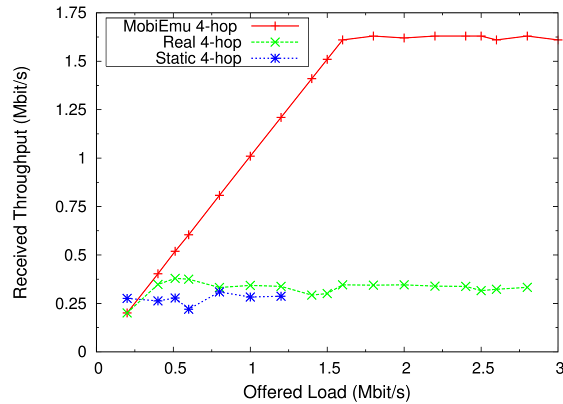


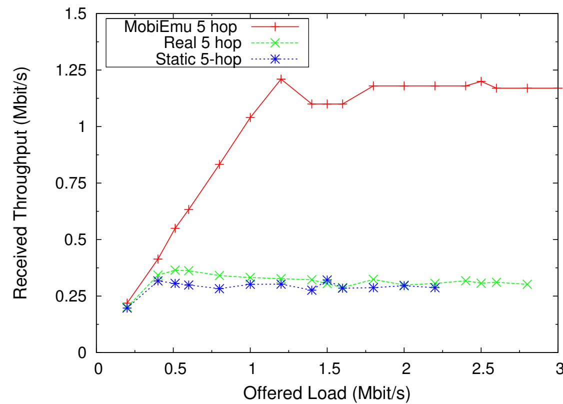
Figure 7.4: Offered load vs. received throughput for the 2-hop and the 3-hop experiments.

Looking at figure 7.4 and figure 7.5 we can see that especially for the 2-hop and 3-hop experiments, depicted in figure 7.4a and 7.4b, the throughput received for manually assigned routes is inferior compared to the results obtained when using DYMO. The poor results obtained with static routes are puzzling and we do not have a sensible explanation for the outcome. Time constraints unfortunately

meant that we were not able to repeat the experiments. If we were to repeat the experiments, we expect to see almost the same throughput when using manually assigned routes and when using DYMO, as there is no reasonable explanation why the static setup should perform worse than when using DYMO.



(a)



(b)

Figure 7.5: Offered load vs. received throughput for the 4-hop and the 5-hop experiments.

When comparing the results obtained for DYMO using the MobiEmu setup and the real setup we see, as expected, that for the MobiEmu setup the maximum received throughput are much better than for the real setup. As explained in section 7.2, in the MobiEmu setup, the nodes are within the same collision domain and does not suffer from the effects of the hidden terminal problem.

7.5 End-to-End Delay

We conducted an experiment to measure the round-trip time (RTT) in the multihop ad hoc network. This experiment was carried out for different packet sizes

and different hop counts on the chain topology. RTT was measured using the *ping* utility with the adaptive option, which adjusts the interpacket interval according to the RTT, so there should never be more than one unanswered ping request in the network at a time.

The packet size used with the ping command was respectively the default value (56 bytes) and 1016 bytes. This amounts to a total of 64 ICMP data bytes or 1024 ICMP data bytes as the length of the ICMP header is 8 bytes. A total of 1024 packets were transmitted.

Table 7.5 shows the observed RTT values in milliseconds. The results show the expected trends: the RTT increases with packet size, and with the number of hops traversed.

Number of hops	Ping Size (bytes)	Min	Max	Average	Std. dev.
1	64	1.62	85.4	2.0	2.6
	1024	4.99	88.8	5.2	2.6
2	64	3.26	438.0	5.0	13.6
	1024	9.74	471.0	20.8	25.2
3	64	5.16	1574.0	50.1	82.6
	1024	13.70	760.0	28.2	36.5
4	64	6.53	2145.0	12.0	75.8
	1024	17.00	1275.0	28.1	45.9
5	64	8.24	2561.0	27.8	97.5
	1024	20.20	1565.0	40.8	61.6

Table 7.5: Measured round-trip times (ms).

To better illustrate the differences in RTT when varying the packet size, we also show the results in figure 7.6. If we first disregard the two 1-hop results and second the 3-hop result with packet size of 64 bytes, which is substantially different compared to the other numbers, the difference in RTT between a packet size of 64 bytes and a packet size of 1024 size is between 13 ms and 16 ms. The differences found by Gupta et al. [GWW04] in the 2, 3, and 4-hop experiments are between 7 ms and 15 ms and thus comparable to our numbers.

To get an estimate of the round-trip time in a network with load, we performed the same experiment while simultaneously downloading a file, using FTP, from the same node we were pinging. The experiments were only conducted with the default packet size and unfortunately, we are missing the results from the 2-hop and 5-hop experiments. Furthermore, as we do not have the full log of the ping output the results presented in table 7.6 are based on a different number of ping packets. The number of packets is shown in the table.

Compared with the end-to-end delay numbers in an unloaded network, delays for the 3-hop and 4-hop experiments are much higher and shows more variability in the data denoted by the higher values of the standard deviation.

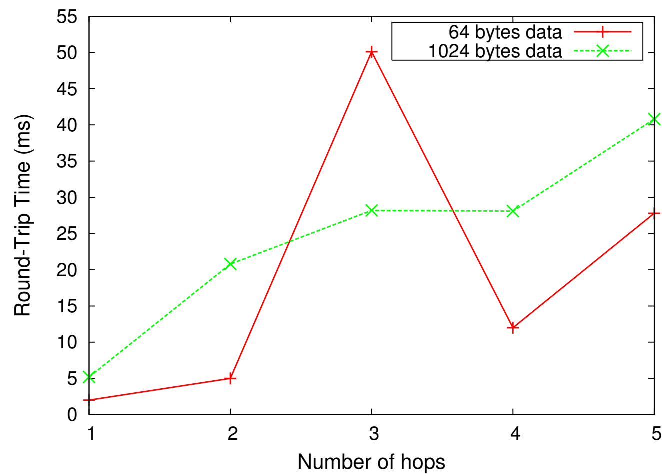


Figure 7.6: Measured round-trip times (ms) for 64 bytes and 1024 bytes ping packets.

Number of hops	Number of packets	Min	Max	Average	Std. dev.
1	217	6.3	432.0	176.5	49.6
3	79	106.0	1351.0	630.1	287.6
4	58	58.1	871.0	404.4	214.0

Table 7.6: Measured round-trip times with a simultaneous FTP session (ms).

7.6 TCP Performance

In this section, we present the results obtained when testing TCP performance. As described in the previous sections, the nodes were set up in a chain. Both the setup with five nodes and the setup with six nodes were used. The experiments have primarily been conducted in the real setup; we unfortunately only have numbers from two experiments from the MobiEmu setup.

Two types of applications were used to measure TCP performance. In the first experiment, the Iperf tool described in section 7.2.1 was used. In the second experiment, FTP software was used. Both types of applications will test the maximum achievable throughput. In addition, the same experiments were conducted with the routes to the laptops being manually assigned instead of set up by the DYMO implementation.

We first present the results obtained using the Iperf network traffic generator in the real setup. In figure 7.7 the obtained results for both routes set up by the DYMO implementation as well as manually assigned routes are shown. Results are shown for both the five-node and the six-node setup.

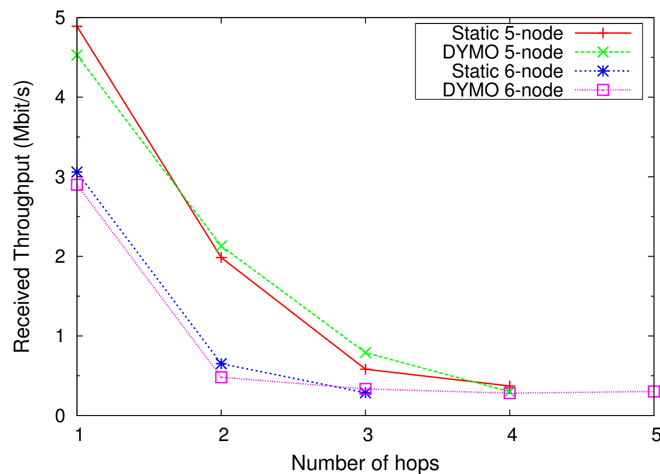


Figure 7.7: TCP Throughput

As mentioned in section 7.2.1 and 7.4, the received throughput in the five-node setup exceeded the throughput received when using all six nodes. As can be seen in figure 7.7, this is also apparent in this experiment. However, at four hops the difference between the results has almost disappeared. The evident disadvantage of using the Dell laptop at longer routes vanished as the contention on the physical layer becomes the major bottleneck.

The obtained results match the results reported by Gupta et al. [GWW04] and Kuladinithi et al. [KUFG] in that larger hop count results in smaller throughput. However, the TCP throughput number Gupta et al. reported for the 4-hop experiment, 1.24 Mbit/s, is quite a bit better than our 0.3 Mbit/s. Kuladinithi et al. also

received higher throughput obtaining approximately 0.75 Mbit/s for a 4-hop experiment. The general observation is that our measured throughput drops more rapidly when the number of hops increase when compared to the results reported by Gupta et al. and Kuladinithi et al.

As mentioned previously in section 7.1, Kuladinithi et al. found that the received throughput using manually assigned routes was slightly lower than when using an AODV implementation to set up routes. Their investigation showed that this was caused by an increased number of duplicate TCP ACKs being generated, which again was caused by smaller values of the TCP retransmission timer compared to when using AODV implementation. More CPU time was used when running AODV and this affected the TCP retransmission timer that again had an influence on the received throughput. Compared to the experiments of Kuladinithi et al., given our numbers, we cannot say anything about the relation between static routes and DYMO assigned routes.

For the MobiEmu setup, we only have numbers for a 5-hop experiment performed with the 6-node setup and the numbers for a 4-hop experiment from the 5-node setup. The results are shown in table 7.7 together with the number obtained from the similar real setup experiments.

Number of hops	Transfer rate (Mbit/s)	
	MobiEmu	Real
4	1.300	0.300
5	1.053	0.302

Table 7.7: TCP throughput in the MobiEmu setup.

As we experienced with the UDP experiments, the received throughput achieved in the MobiEmu setup is superior to that of the real setup. For the 4-hop and 5-hop experiments, the received throughput in the real setup is 23 % and 28 % of received throughput in the MobiEmu setup. The difference in throughput obtained in a lab setup (MobiEmu) and a real setup, is comparable to results reported by Maltz et al. [MBJ99] who in a two-hop TCP scenario measured a difference of 25 % in received throughput between an outdoor setup and an indoor lab setup.

7.6.1 FTP Performance

In this experiment, only the five-node setup was used. Node 2 (see figure 7.1) acts as a server and the last node in the chain initiates a single FTP file transfer. The experiment was repeated with different end-nodes, thus varying the number of hops between one and four. The size of the file to transfer was 13 MB. To be able to trace the transfer progress the tool *tcpdump* [SFR03] was used to capture TCP packets. The experiments were conducted with both DYMO and manually assigned routes.

Figure 7.8 shows the progress at the destination node for the 1-hop and 2-hop experiments for DYMO routes and static routes, respectively. Figure 7.9 shows the similar results for the 3-hop and 4-hop experiments.

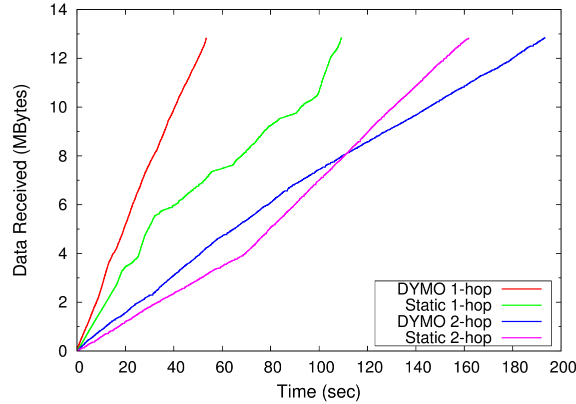


Figure 7.8: FTP Progress with time for 1-hop and 2-hop with DYMO and static routes, respectively. The received bandwidth is 1923, 940, 531, and 635 Kbit/s.

As we experienced in the previous section, we cannot conclude anything with regards to DYMO vs. manually assigned routes. In the 1-hop experiment, it takes twice as long when using the manually assigned routes compared to when DYMO is used. However, the progress for the manual case is uneven, leading to believe that the transfer was in some way obstructed on the physical layer. With two hops, the DYMO routed experiment begins with the fastest throughput, but is then outpaced by the experiment with static routes that finishes approximately 15 seconds before.

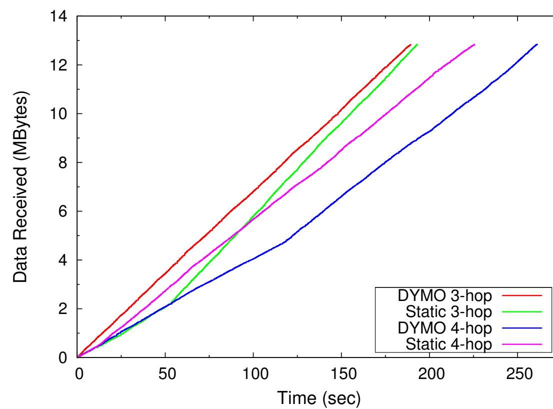


Figure 7.9: FTP Progress with time for 3-hop and 4-hop with DYMO and static routes, respectively. The received bandwidth is 542, 531, 393, and 455 Kbit/s.

In the 3-hop experiment, the DYMO experiment starts best, but after 50 seconds the progress made in the manual experiment makes both transfers end at about the same time. In the 4-hop experiment, the manual experiment ends 35–40 seconds before the DYMO experiment because of better progress made at the beginning of the transfer, but after about 150 seconds, the progress rate is about the same in both experiments.

7.7 Ad Hoc Horizon

As mentioned in section 7.1, Tschudin and Osipov have investigated the usefulness of TCP based network services running on top of IEEE 802.11 networks in which routing is controlled by a MANET routing protocol. The metrics that are used to measure usefulness are the no-progress ratio and the unfairness index. The *no-progress ratio* is the relation between time intervals larger than three seconds in which TCP does not make progress and the total duration of the TCP session. The *unfairness* u among TCP sessions is defined as, where f is *Jain's fairness index* [JCH84],

$$f = \frac{(\sum_{i=1}^{n} x_i)^2}{n \sum_{i=1}^{n} x_i^2}, \qquad u = 1 - f$$

and x_i is the received throughput of FTP session i. In our calculations, we have used the size of the received files at end of experiments as a measure of the received throughput. The fairness index is bounded between 0 and 1; if the bandwidth is equally partitioned between the sessions, the index is 1. If k of the n sessions receive equal bandwidth, and the others get none, the index is k/n. The fairness index f is complemented to get an unfairness index u.

The experiments are performed in a set of topologies named *beam star* network topologies. A central node acts as a server and this server is the origin of paths or beams that end at nodes distanced an identical number of hops away. Three examples of beam start topologies are depicted in figure 7.10. To be able to distinguish between the members of the beam star family we say an *AxB* beam star topology has *A* beams and *B* number of hops from the central node to an end-node.

We have performed some experiments to evaluate the ad hoc horizon for a small subset of the topologies tested by Tschudin and Osipov. The limiting factor is the number of laptops available. For the experiments requiring seven nodes one additional laptop was used: a Toshiba Satellite Pro laptop with a 1 GHz Pentium III Mobile processor, 640 MB of RAM, and a D-Link DWL-660 PCMCIA 802.11b wireless adapter. The tested topologies are primarily one-hop topologies. In addition, the topologies 2x3 and 2x2 have been tested. It was planned to also test 3x2, unfortunately, we missed this test during the experiments. The 2x1, 2x2, and 2x3 experiments were conducted in the real setup and the rest, the (3-6)x1 were performed using the MobiEmu setup because of the difficulties of setting up these beam star topologies indoors.

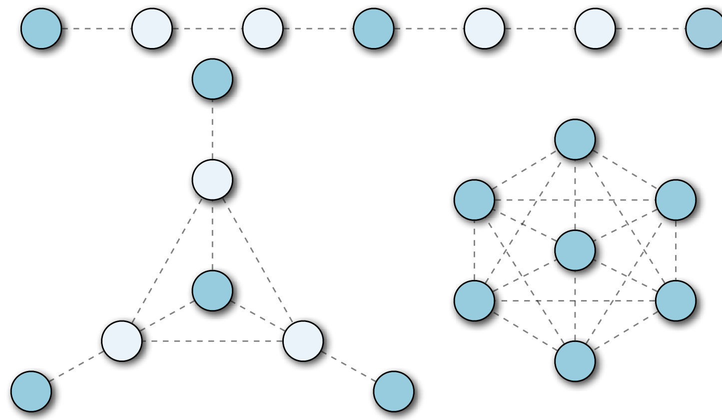


Figure 7.10: The beam star network topologies, 2x3, 3x2, and 6x1.

7.7.1 Measuring TCP Unfairness

Compared to the setup of Tschudin and Osipov in which the central node initiate the FTP session, in our setup, the end-nodes begin the FTP transfer. Each transfer starts randomly within 5 seconds when the experiment begins. We do not have any numbers for the no-progress ratio, but informal observations during the tests did not reveal any noticeable stalls during the FTP transfer sessions. The results obtained for the TCP unfairness are illustrated in figure 7.11.

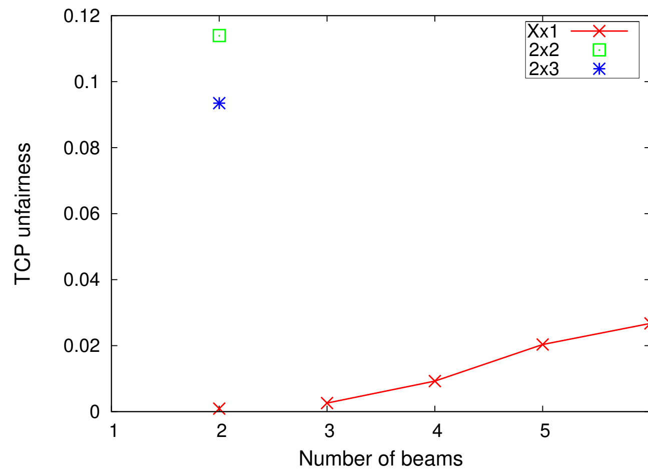


Figure 7.11: TCP unfairness. The 2x1, 2x2, and 2x3 experiments have been conducted in the real setup. The (3-6)x1 experiments have been conducted in the MobiEmu setup.

To distinguish which experiments have been performed in the real setup and which are performed in the MobiEmu setup, in the figure, the TCP unfairness

number for the 2x1 experiment is not connected to rest of the one hop experiments, which have been linked together with a line.

In the figure, we see that although the 2x1 experiment has also been conducted in the real setup, the unfairness number is much lower than that of the 2x2 and 2x3 experiments. The number compares with the other one hop experiments that were conducted in the MobiEmu setup. We have the following explanation for this: When both FTP sessions have been initiated and are up and running, it is the central node, the FTP server that transmits data to the two end-nodes. The data transmission to the two nodes is therefore not simultaneously, but is coordinated by the FTP server. Only the TCP ACK replies sent by the end-nodes are uncoordinated and can collide at the central node (the hidden terminal problem). With the assumption that the FTP data packets outnumber the TCP ACKs, this increases the reliability of both links. Furthermore, the packets of the FTP sessions only travel one hop, compared to the multiple hops of the 2x2 and 2x3 experiments.

Tschudin and Osipov ran simulation with an AODV implementation both with and without link layer feedback. Tschudin and Osipov found that the no-progress ratio for the 2x3 experiment was unexpectedly high even with link layer feedback enabled. Analysing the two sessions they discovered the sessions would take turns in starving each other. At one point, one of the sessions had a no-progress period of 40 seconds. Despite the rather large no-progress ratio of the experiment, the unfairness number was relatively low. Our observed unfairness for the 2x3 experiment compares well with the one reported by Tschudin and Osipov for their simulations with no link layer feedback enabled. Generally, our results confirm the observations made by Tschudin and Osipov namely that TCP performance stated in the terms of TCP unfairness starts degrading after two hops. With regards to the number of beams, Tschudin and Osipov only report TCP unfairness up to four beams with numbers matching ours. If the obtained numbers for the experiments conducted in the MobiEmu setup are realistic, at least for the one hop experiments adding additional beams, the TCP unfairness is to a large degree unaffected.

Given that most of the 1-hop topology experiments have been conducted in the MobiEmu setup, it is interesting to estimate how the results compare to hypothetical results obtained in the real setup. Using the 6x1 experiment as an example, in the real setup, of the six end-nodes, an arbitrary end-node will be able to hear all but one of the other end-nodes. In the MobiEmu setup, an end-node can hear all the other end-nodes. As this is not a major difference, we conclude that the results are comparable even given our experience with the TCP experiments and the difference between the two setups described in section 7.2.

7.7.2 Measuring HTTP Download Times

For the HTTP experiments, only one route discovery cycle was performed, i.e., overall three RREQ were transmitted. The second RREQ is transmitted one second after the first RREQ and the third RREQ is transmitted three seconds after the first RREQ. After the third RREQ has been transmitted, the node waits four

seconds before ending the route discovery cycle. Thus, if no RREP has been received during this period, a total of seven seconds will have elapsed before the route discovery cycle ends.

For the HTTP experiments with the three real topologies 2x1, 2x2, and 2x3, a large part of the session ended without the requesting node ever finding a route to the server node. The success ratios in percent were 57.58 %, 66.67 %, and 64.29 %, respectively. To be able to present a number for these cases we have calculated the average of the successful HTTP session and added 7 seconds, which is the delay before unsuccessful route discovery session times out, as explained above. This implicitly means we assume a success ratio of 100 % for the next route discovery attempt, which of course is not realistic. So if more than one route discovery cycle had been used, the download times might have been even higher.

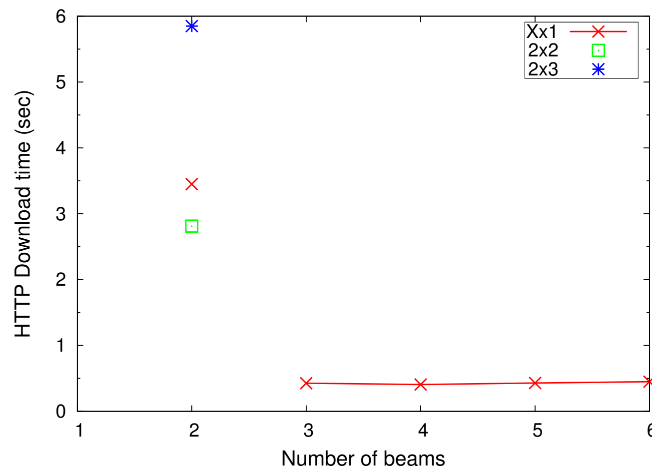


Figure 7.12: Download times for HTTP sessions competing with a single FTP session in some of the beam star scenarios.

Once again to distinguish the experiments, the HTTP download time for the *2x1* experiment is not connected to rest of the one hop experiments that have been linked together with a line.

Compared to the experiments measuring the TCP unfairness there is a large difference how the two 2x1 experiments compare. In the TCP unfairness experiments, the 2x1 experiment compare very well to the (3-6)x1 experiments, i.e., the unfairness is lower than for the (3-6)x1 experiments, even though the first is conducted in the real setup and the second ones are conducted in the MobiEmu setup. This is not the case for the HTTP download time experiments where the obtained time for the 2x1 experiment is over 7 times higher than that for the (3-6)x1 experiments.

The difference between the two types of experiments is that when measuring TCP unfairness, both transfers are started within a five second interval. In the HTTP download experiment, when end-node 1 requests the web page, the FTP session to end-node 2 is already established and running. Node 1 then has to com-

pete with the already running session. Additionally, as previously mentioned, in the FTP experiment the central node acts as a coordinator of the FTP data packets, which means that only TCP ACK packets sent by end-nodes are uncoordinated and are likely to collide at the central node. Thus, neither of the two nodes has any advantage compared to the other. In the HTTP download experiment, even though end-node 1 can sense when the central node occupies the link, it cannot hear the other end-node and packets can collide at the central node rendering the node unable to establish the HTTP session.

Comparing to the results of Tschudin and Osipov, for the small number of experiments our results confirm that multi-hop topologies influence performance, in this case download times.

7.8 Experiences Learned

Placing the laptops in an indoor setting to form five-hop chain topology proved to be a major challenge. We experienced that even when laptops were placed in opposite ends of long hallways, if there were no major obstacles they could still successfully deliver packets to each other. On the other side, placing a laptop a few meters into an office in the hallway or around corners could hamper the transmission to a laptop not distanced far away. So it was difficult to position laptops to build a chain topology; laptops had to be to carefully placed around corners and in offices. One should expect to make several attempts before having a reproducible and reliable setup. Similar experiences where reported by Kiess et al. [KZTM05] and Desilva and Das [DD00]

Further to the difficulties of setting up laptops in a chain, we also experienced what Lundgren et al. call communication gray zones [LNT02]. In a communication gray zone, a node experiences significant loss of unicast packets while having little or no problem when receiving broadcast traffic. Lundgren et al. experienced that nodes reported to be neighbours based on the exchange of Hello messages (which are sent using broadcast) would fail to receive unicast data packets. One of the major causes of communication gray zones is difference in transmission rates. In IEEE 802.11b, broadcasting is done at a basic rate to ensure backwards compatibility with IEEE 802.11 while data transmissions normally are sent at higher rates. In IEEE 802.11b this up to 11 Mbit/s. The slower transmission rate for broadcast traffic makes packets transmitted this way reach further.

The conclusion of the investigations of Lundgren et al. was that the communication gray zones “leads to invalid routing table entries for protocols that establish their neighbor set using Hello beacons” [LNT02]. Our implementation does not use Hello message, however, we experienced problems similar to Lundgren et al. During our experiments with setting up the chain topology, we found that sometimes it was difficult to establish 2-hop routes if the two laptops distanced furthest apart were too close to each other. With the communication gray zone problem in mind, the following course of events were assumed and later validated by inspect-

ing the routing daemon log file of the various nodes. The example is illustrated in figure 7.13.

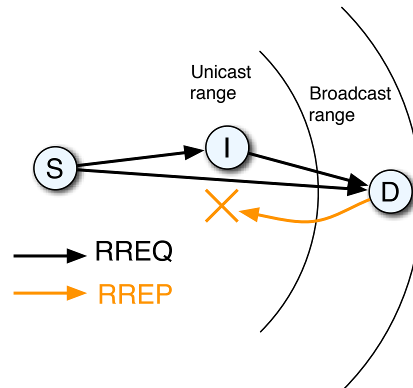


Figure 7.13: Communication gray zone problem when discovering routes.

When the node **S** initiates route discovery to find a route to node **D**, it broadcasts an RREQ. The RREQ is received by both **D** and the intermediate node **I**. When **I** receives the RREQ it appends its own address to the RREQ and resends the RREQ. When **D** receives the RREQ it processes the packet and creates an RREP to answer **S** as it is the target of the RREQ. The RREP message sent to answer the RREQ from **S** is never received by **S** as it is not within the unicast transmission range of **D**. When **D** immediately after receives the RREQ from **I**, it discards the packet, as it has already processed an RREQ with the same sequence number. Consequently, no route is ever established between **S** and **D**.

Besides the difference in transmission rate, Lundgren et al. also mention the difference in packet size as a contributing factor to the communication gray zones problem. In their setup, they observed problems with ordinary data packets as opposed to Hello messages. Data packets are usually much larger than Hello messages, which have a length of 20 bytes and the probability of reception is decreased as the packet is more likely to be damaged in transit. This observation was confirmed by Belding-Royer and Chakeres [CBR02] who did experiments to determine the influence of packet size on the packet delivery rate and found that small packets are more likely to be received than large packets.

In our example, the packets either broadcasted or unicasted were identical in size, so the size factor was not contributing to the communication gray zone effects we experienced.

8
Conclusions and Future Work

In this chapter, we summarize and conclude this thesis and give directions for future work.

8.1 Summary

The DYMO routing protocol is a newly proposed on-demand MANET routing protocol. It is currently defined in an Internet-Draft in its sixth version and is thus, work in progress.

We introduced MANETs, the envisioned application areas, and the challenges imposed on MANET routing protocols. Routing in a MANET must be efficient in a broad set of imaginable scenarios and the two categories of MANET routing protocols were introduced; on-demand/reactive protocols and table-driven/proactive protocols. We presented the DYMO routing protocol in detail and traced its basic set of operations primarily to AODV, but also to DSR. In addition, two proposed modified versions of the AODV protocol; AODVjr and AODV-PA, were mentioned to have had impact on the DYMO protocol draft.

8.1.1 Implementation

Our literature survey has identified the challenges of implementing on-demand MANET routing protocols caused by the routing architecture in current operating systems. Because the challenges covered a broad problem domain, we identified the ones pertinent for the further discussion. For example, one of the challenges was to determine when to initiate route discovery. Further to the identification of the implementation challenges we presented the following implementation techniques to meet the challenges: the kernel modification approach, the snooping approach, and the netfilter approach. The emphasis was on solutions for Linux.

Based on the presented implementation approaches we developed an implementation of the DYMO routing protocol. We then described the design of the implementation as well as the structure of the code. The implementation is based

on the netfilter approach in which user-defined code can be inserted into the Linux network stack and alter the flow of packets. The implementation consists of a user space daemon and an accompanying Linux kernel module and the implementation is written in C and in the scripting language Lua. The kernel module implements the netfilter functionality, active route management, and communication with the user space daemon. We were interested in reducing the traffic exchanged between the Linux kernel and our routing daemon while still partly keeping route timeouts updated in the routing daemon and presented three methods to accomplish this task. We were interested in a portable implementation and have isolated operating system specific code.

Netfilter The use of the netfilter framework made it possible to develop an efficient implementation of the DYMO protocol. Data packets are not required to be copied from kernel space to user space and back which is necessary when using the snooping approach. In addition, the use of netfilter and a Linux kernel module makes it possible to avoid communication between user space and kernel space when a route has been established and consequently eliminate routing overhead.

Compared to the snooping approach, the use of netfilter complicates the process of porting our DYMO implementation to other operating systems because of the required kernel module. Programming at the kernel level is always more difficult than at the user level. Furthermore, the kernel of other operating systems may not provide the same functionality as netfilter which can make it necessary to modify the user space routing daemon.

Lua The use of the Lua dynamic language made the implementation progress process faster and easier. Lua provides a layer of abstraction above the one provided by the low-level C programming language which allowed us to implement the DYMO protocol in fewer lines of code and in a garbage collected environment with no need for the tedious and error-prone memory management required by C. The use of Lua made the implementation less dependent on the binary representation of DYMO control messages which allowed us to update the implementation to use a revised DYMO message layout without a total rewrite of the protocol logic layer.

In many of our implementation design choices, we focused on performance. Making the choice of writing parts of the implementation in Lua, we traded-off performance for flexibility. While the DYMO protocol specification is still work in progress, in this case we favour flexibility over performance. However, we made sure the individual parts of the implementation can be rewritten in C, which adds memory management complexity. For example, in C, we cannot simply delete an object passed from Lua, as there can still exist references to the object in Lua. Thus, while we regard the choice of Lua for the implementation as a success, the added flexibility and the emphasis on language interchangeability added complexity to the implementation.

8.1.2 Experimental Evaluation

Experimental evaluation made in the literature suggested a set of practical experiments to conduct. Because this thesis is a one-person-project, a simple chain topology was chosen in nearly every case. Most experiments were conducted in a setup in which laptop computers where placed properly distanced in hallways, as well as in a setup using the evaluation testbed software MobiEmu. We performed practical experiments to investigate:

1. Route Discovery Latency. In the real setup, the route discovery latency was found to between around 50 ms for discovering a node 1 hop away and a little more than 800 ms for a node 5 hops away. We compared the obtained numbers with results from similar experiments with AODV implementations and found them to compare well to an experiment using an implementation resembling our own. The MobiEmu setup experiment generally showed lower latencies, but differences were less noticeable compared to the UDP and TCP throughput experiments.

2. UDP Throughput. We tested UDP throughput in the real setup, the MobiEmu setup, and in a setup with manually assigned routing table routes. We tested received throughput vs. offered load. Generally offered load ranging from 0.2 Mbit/s to 3 Mbit/s was tested. In all three setups, the received throughput would stabilize at a fixed level even with an increased offered load. We found the received throughput in the MobiEmu setup to be three to four times higher than that of the real and manual setup.

3. End-to-End Delay. We found the end-to-end delay to increase with the number of hops and with packet sizes as expected. We compared the results to similar experiments and found that the difference in end-to-end delay in experiments with various packet sizes was similar to our obtained results.

4. TCP Throughput and FTP Performance. As expected and in accordance with previous results, in both experiments a larger hop count resulted in smaller throughput. In the TCP throughput experiment, the throughput we received for the 3-hop and 4-hop experiment were considerably lower when compared to similar results. Both type of experiments were also conducted with manually assigned routes. Because of inconclusive results, no conclusion regarding the advantage of DYMO assigned routes or manually assigned routes were made. Two of the TCP throughput experiments were conducted in the MobiEmu setup. We found the received throughput of the corresponding experiments in the real setup to be around 25 % of that of the MobiEmu setup.

5. Unfairness Ratio and Download Time (beam star setup). Experiments in the set of beam star topologies had previously only been conducted using

simulation. We conducted practical experiments and our obtained results were found to be similar to the simulation results.

We became increasingly experienced while conducting practical experiments but we only had access to laptop computers for a limited period. This means that we did not have the opportunity to repeat some of the experiments that were incomplete or gave curious results, typically by deviating notably from results obtained from comparable experiments.

8.2 Conclusions

In the following, we list the findings we have made because of our work.

8.2.1 Implementation

As a part of this thesis project, we explored the integrated use of a scripting language in a C-based implementation and explored route table timeout strategies. Our implementation has to a large degree been designed based on the experience documented by others. By exploring implementation possibilities and describing our design, we extend the possible ways with which a MANET on-demand routing protocol can be implemented as we allow others to benefit from our experiences. This in return provide for a more efficient implementation process and implementations that are more efficient.

As a part of the implementation process, we have made a review of the DYMO Internet-Draft in order to implement it. As described in section 5.5.1, we found a couple of errors and ambiguities.

With regards to the preparation of MANET Internet-Drafts, specifically the DYMO Internet-Draft we have some remarks based on our experience in implementing the DYMO specification. We find that it would be useful if the protocol author produces a document containing annotations to go along with a main draft document. Most specifications are terse by nature and it can sometimes be a challenge to interpret what a statement such as “the result of subtracting Route.SeqNum from Node.SeqNum is less than or equal to zero” means in terms of the protocol design decision. First, one has to rephrase it: “the sequence number of the DYMO message is newer than (or identical to) the sequence number found at the node”. Second, one must understand why this must be the case for the specific incident. This is often easy, but not in all cases. We do not believe it would desirable to include annotations into the main draft document. From our point of view (someone who implemented the DYMO protocol), one of the strengths of the DYMO specification is that it is short and written with protocol implementers in mind. However, as stated above, there is a trade-off between conciseness and the effort that must be investigated to understand the specification in the first place.

8.2.2 Experimental Evaluation

We conducted one of the first experimental evaluations of the DYMO protocol. The results enable others to use our work as a baseline for comparison when performing their own evaluation. The experiments provided practical evidence that throughput in a MANET routed with DYMO is similar to throughput in a MANET routed with AODV.

We compared results obtained in a emulated setup with corresponding real world setup. The findings allow one to give an estimate on expected real world throughput based on emulated results. This can be useful when real practical experiments are impossible or infeasible because of some constraints or simply when it is desirable to give an estimate of real world performance during the implementation process.

We performed the first practical experiments to estimate the ad hoc horizon which is intended to capture the limit on the number of hops and number of nodes when TCP based network services stop being useful in a MANET. Experiments involving the ad hoc horizon are important because they try to take into account the usage patterns of everyday use like web browsing with a concurrent long-standing download session. Our small-scale experiments confirmed the limits previously reported using simulation and furthermore show that experiments evaluating the practically applicability of MANETs are important.

8.3 Future Work and Research

In this section, we give an outline of future work and research. First, we outline future work with regards to the developed DYMO implementation. Among others, we describe the work required to update the implementation to comply with the sixth version of the Internet-Draft. Second, we outline future work with regards to the experimental evaluation.

8.3.1 Implementation

In this section, we describe future work with regards to our DYMO implementation. We first describe which of the existing parts of the implementation that have to be changed in order to update it to conform to the sixth version of the Internet-Draft. We then describe some issues regarding performance of the implementation and how to possibly improve it. Finally, we discuss portability and what is required to port our DYMO implementation to Mac OS X.

Updating the Implementation to the DYMO Draft Version Six Our implementation of the DYMO specification complies partly with the fourth version of the draft. The limitations of our implementation compared to the fourth version of the DYMO draft were listed in section 5.5.2. All the listed limitations will also

have to be implemented in order to comply with the sixth version of the DYMO draft.

One of the listed limitations was the lack of support for the MANET Neighborhood Discovery Protocol (NHDP) [CDD06b]. The support for NHDP is not mandatory, but we predict support to be important in order for the DYMO-AU implementation to be interoperable with current and future DYMO implementations. We also mention it here, as the implementation task is considerable: the NHDP Internet-Draft is complex, introduces a set of new concepts not found in the DYMO specification, and is longer than the DYMO Internet-Draft.

Aside from the task of supporting NHDP, the primary change between version four and version six of the DYMO draft is that the sixth version has been updated to use a newer version of the *generalized MANET packet and message format*, specifically version two [CDDA06]. This version of the packet format sees a couple of major updates compared to the packet format used in the fourth version of the DYMO draft, which was version zero. This means that the parts of code dealing with reading and writing of packet in this format must be updated.

The second major change that must be made to the implementation is the implementation of timeouts. In the fourth version of the draft, two kinds of route table entry timeouts are specified. The Valid Timeout specifies when a route table entry becomes invalid and the Delete Timeout specifies when it should be deleted. In the sixth version, the number of timeouts has been extended to five. They are

- Minimum Delete Timeout
- Maximum Delete Timeout
- New Information Timeout
- Recently Used Timeout
- Delete Information Timeout

We do not go into details of the semantics of the various timeouts, but just note that the parts of the implementation dealing with timeouts will have to be rewritten.

Performance No parts of our implementation have at this time been tested with respect to performance. For example, in section 6.1.1 we mentioned that the timer queue had been implemented using a priority queue. As the practical experiments we conducted involved a maximum of seven nodes, it is likely that a linear search in an array is faster. However, following the common software advice of not optimizing prematurely, we have not yet made any modifications to the working code. Profiling the program will tell if any changes will have any noticeable effects on performance.

In section 6.3, we presented three possible methods for updating the timeout value in the routing daemon. The version that is currently implemented has the

drawback that a message must be transferred from the kernel module to the routing daemon whenever the node sends, receives, or forwards a packet.

Because of their advantages, it is clear that the two approaches that have not yet been implemented warrant further investigations. We find the *on-demand update of timeouts* proposal interesting because few messages cross the kernel/user space boundary.

With regards to the implementation of the two other proposals we developed an initial implementation of the on-demand approach described in section 6.3.3, albeit without the delete timeout garbage collection timer task safety measure. Initial experiments using 2-3 nodes showed it to function properly, however, we did not perform any large-scale experiments to ensure conformance to the specification and neither have we conducted any performance analysis.

Portability In section 5.5.3, we claimed that the DYMO-AU implementation has been designed with portability in mind. As previously mentioned, the current implementation only supports Linux. We now give a short outline of the practical challenge of porting the implementation to another operating system, using Mac OS X as an example.

Code using network sockets are to a large extent directly portable across any POSIX compliant operating system. However, the various options that can be set on open sockets with the `setsockopt` [SFR03] call differ between various systems. For example, it is necessary to know the source address and receiving interface when processing a DYMO message. On BSD derived systems, the socket options IP_RECVDSTADDR and IP_RECVIF are available,[1] while the IP_PKTINFO option is available on Linux and cover both cases. To allow for future version of the daemon running on BSD derived systems, both methods have been implemented.

Because we have considered BSD derived systems during the implementation process, the routing daemon is to a large degree already directly portable to Mac OS X. The major challenge is the kernel module. On Mac OS X, a *Network Kernel Extension* (NKE) [App05] is roughly the equivalent of a Linux Kernel Module. It is necessary to write an NKE that implements the same functionality provided by the Linux kernel module. Mac OS X has support for three kinds of network filters, called *Socket*, *IP*, and *Interface* filters, respectively that can manipulate the network traffic at various levels of the network protocol stack. A communication interface similar to netlink sockets for interaction between kernel extensions and users space applications is available. We still have to investigate how easily the design of the Linux kernel module can be translated to a Mac OS X NKE and the various network filters.

[1] FreeBSD, NetBSD, and OpenBSD are examples of BSD derived systems. Mac OS X contains code from a BSD derived system.

8.3.2 Practical Evaluation

In section 7.2.1, we mentioned that the RTS/CTS clearing procedure was disabled during our experiments. It could be interesting to repeat the conducted experiments with the RTS/CTS clearing procedure enabled and compare the results with the ones described in this thesis in order to examine the effect of the hidden terminal problem.

Our DYMO implementation does not implement any form of neighbour discovery. Consequently, when a route has been established there is no routing protocol overhead on the link layer, i.e., no control messages are transmitted while a route is still active. In could be interesting to repeat the conducted experiments with an implementation supporting some sort of neighbour discovery, for example, NHDP, in order to compare to the results described in this thesis.

Finally, in our evaluation, some of the experiments have not been conducted for all the envisioned number of hops and scenarios. We think, it is of value to repeat the chosen set of experiments to get a better basis of comparison.

Interoperability We have several times mentioned that independently developed implementation of a routing protocol defined in an IETF Internet-Draft must be shown to be interoperable before the draft can be promoted to an RFC. Two other implementations of the DYMO protocol are available. DYMOUM [RR] conforms to the fifth version of the draft [CP06c], however, it does not use the specified generalized MANET packet format, but the old packet format specified in the DYMO draft prior to version four. Similar, the NIST DYMO implementation has not been updated since the release of the second DYMO draft. This makes it so far impossible to carry out an interoperability evaluation, and this is why no interoperability evaluation has been conducted as part of this thesis. However, as described, interoperability between implementations is important and interoperability experiments should be conducted at some point.

Setting Linux Kernel Parameters

In order for the DYMO-AU implementation to work flawlessly on Linux, a couple of kernel parameters must be modified. As explained in section 6.2.2, kernel parameters can be modified by writing to files in the /proc files system. The most important variable is */proc/sys/net/ipv4/ip_forward*, which must be set to *1* to allow the local host to forward IP packets and operate as a router.

Second, as the DYMO routing daemon is responsible for routing on interfaces enabled for DYMO operation, no ICMP redirect messages should be sent or accepted by the local host. Thus, sending or acceptance of redirect messages are disabled.

Finally, to have newly installed routes take effect immediately, the forwarding cache (see section 4.1) used by the Linux kernel forwarding function must be flushed. However, there is a delay between a new route has been installed in the kernel routing table and the forwarding cache is flushed. On the Ubuntu 5.10 Linux distribution with the default settings, the minimum delay is 2 seconds and the maximum delay is 10 seconds. These values have been set to *0*.

B
Contents of the CD-ROM

This report is accompanied by a CD-ROM including the source code of the DYMO-AU implementation of the DYMO routing protocol. Instruction on how to build the implementation can be found in the source code directory. This report can also be found on the CD-ROM. The CD-ROM is organized in the following directories:

`dymo-au/src/` contains the part of the implementation written in C

`dymo-au/src/lua/` contains the part of the implementation written in Lua

`dymo-au/src/linux/` contains the Linux specific code. The Linux kernel module code is located in this directory

`dymo-au/lua-5.0.3/` contains the Lua distribution. To ease the build process it is included together with the DYMO-AU source code and can optionally be built and used together with DYMO-AU.

`dymo-au/tolua++-1.0.92/` contains the source for the tool used to generate Lua-C glue code. To ease the build process it is included together with the DYMO-AU and Lua source code and can optionally be built and used together with DYMO-AU and the provided Lua distribution.

`report/` contains this report.

`articles/` contains unpublished articles referenced in this report.

References

[Adh] Ad hoc routing protocol list. http://en.wikipedia.org/wiki/Ad_hoc_routing_protocol_list. Last accessed November 2006.

[AGSI02] Jeremie Allard, Paul Gonin, Minoo Singh, and Golden G. Richard III. A user level framework for ad hoc routing. In *LCN '02: Proceedings of the 27th Annual IEEE Conference on Local Computer Networks*, page 13, Washington, DC, USA, November 2002. IEEE.

[App05] Network kernel extension programming guide. http://developer.apple.com/documentation/Darwin/Conceptual/NKEConceptual/, August 2005.

[Ara] ARAN main page. http://prisms.cs.umass.edu/arand/. Last accessed December 2006.

[Bal] Pixel Ballistics. LuaObjCBridge. http://www.pixelballistics.com/Software/LuaObjCBridge/Contents.html. Last accessed September 2006.

[BEF+00] Lee Breslau, Deborah Estrin, Kevin Fall, Sally Floyd, John Heidemann, Ahmed Helmy, Polly Huang, Steven McCanne, Kannan Varadhan, Ya Xu, and Haobo Yu. Advances in network simulation. *Computer*, 33(5):59–67, 2000.

[BLG00] Sang Ho Bae, Sung-Ju Lee, and Mario Gerla. Unicast performance analysis of the ODMRP in a mobile ad hoc network testbed. In *Proceedings of the Ninth International Conference on Computer Communications and Networks*, pages 148–153, Las Vegas, NV, USA, October 2000. IEEE.

[Bra96] Scott O. Bradner. The internet standards process – revision 3. RFC 2026, IETF, October 1996. rfc2026.txt.

[BRCJP04] Elizabeth Belding-Royer, Ian Chakeres, David Johnson, and Charlie Perkins. DyMO – dynamic MANET on-demand routing protocol. In Rebecca Bunch, editor, *Proceedings of the Sixty-First Internet Engineering Task Force*, Washington, DC, USA, November 2004. IETF.

[Bro03] Martin A. Brown. Guide to IP layer network administration with linux. http://linux-ip.net/, April 2003.

[Byt] Bytecode compiler benchmark. http://unigine.com/products/unigine_v0.32/compiler_benchmark. Last accessed September 2006.

[Car03] Christopher Cardé. An interaction model for decoupled implementation and evaluation of mobile ad-hoc routing protocols. http://www.carde.com/static/documents/research/decoupled-ad-hoc-routing/decoupled-ad-hoc-routing-s03-carde.pdf, 2003.

[CBR02] Ian D. Chakeres and Elizabeth M. Belding-Royer. The utility of hello messages for determining link connectivity. In *Proceedings of the 5th International Symposium of Wireless Personal Multimedia Communications (WPMC) 2002*, volume 2, pages 504–508, Honolulu, Hawaii, October 2002.

[CBR04] Ian D. Chakeres and Elizabeth M. Belding-Royer. AODV routing protocol implementation design. In *ICDCSW '04: Proceedings of the 24th International Conference on Distributed Computing Systems Workshops - W7: EC (ICDCSW'04)*, pages 698–703, Washington, DC, USA, 2004. IEEE.

[CBR05] Ian D. Chakeres and Elizabeth M. Belding-Royer. AODV implementation design and performance evaluation. *International Journal of Wireless and Mobile Computing (IJWMC)*, 2/3, 2005.

[CDD06a] Thomas Clausen, Christopher Dearlove, and Justin Dean. Generalized MANET packet/message format. Internet-Draft Version 0, IETF, February 2006. draft-ietf-manet-packetbb-00.txt, (Work in Progress).

[CDD06b] Thomas Clausen, Christopher Dearlove, and Justin Dean. The MANET neighborhood discovery protocol (nhdp). Internet-Draft Version 0, IETF, June 2006. draft-ietf-manet-nhdp-00.txt, (Work in Progress).

[CDDA06] Thomas Clausen, Christopher Dearlove, Justin Dean, and Cedric Adjih. Generalized MANET packet/message format. Internet-Draft Version 2, IETF, July 2006. draft-ietf-manet-packetbb-02.txt, (Work in Progress).

[CJ03] Thomas Clausen and Philippe Jacquet. Optimized link state routing protocol (OLSR). RFC 3626, IETF, October 2003. rfc3626.txt.

[CJ06] Thomas Clausen and Philippe Jacquet. The Optimized Link State Routing Protocol version 2. Internet-Draft Version 2, IETF, June 2006. draft-ietf-manet-olsrv2-02.txt, (Work in Progress).

[CJWK02] Kwan-Wu Chin, John Judge, Aidan Williams, and Roger Kermode. Implementation experience with MANET routing protocols. *ACM SIGCOMM Computer Communication Review*, 32(5):49–59, November 2002.

[CKB02] Ian D. Chakeres and Luke Klein-Berndt. AODVjr, AODV simplified. *ACM SIGMOBILE Mobile Computing Communications Review*, 6(3):100–101, July 2002.

[CKG+] Patrick Charles, Dave Knoester, John Guthrie, Justin Haddad, and Steve Bitteker. Network packet capture facility for java. `http://jpcap.sourceforge.net/`.

[CM03] Carlos Miguel Tavares Calafate and Pietro Manzoni. A multi-platform programming interface for protocol development. In *Eleventh Euromicro Conference on Parallel, Distributed and Network-Based Processing (Euro-PDP'03)*, page 243, February 2003.

[Com00] Douglas E. Comer. *Internetworking with TCP/IP: Principles, Protocols, and Architecture*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, fourth edition, 2000.

[CP06a] Ian D. Chakeres and Charles E. Perkins. Dynamic MANET on-demand (DYMO) routing protocol. Internet-Draft Version 6, IETF, October 2006. draft-ietf-manet-dymo-06.txt, (Work in Progress).

[CP06b] Ian D. Chakeres and Charles E. Perkins. Dynamic MANET on-demand (DYMO) routing protocol. Internet-Draft Version 4, IETF, March 2006. draft-ietf-manet-dymo-04.txt, (Work in Progress).

[CP06c] Ian D. Chakeres and Charles E. Perkins. Dynamic MANET on-demand (DYMO) routing protocol. Internet-Draft Version 5, IETF, June 2006. draft-ietf-manet-dymo-05.txt, (Work in Progress).

[DD00] Saman Desilva and Samir R. Das. Experimental evaluation of a wireless ad hoc network. In *Proceedings of the 9th Int. Conf. on Computer Communications and Networks (IC3N)*, pages 528–534, Las Vegas, NV, USA, October 2000.

[Fow03] Martin Fowler. *UML Distilled: A Brief Guide to the Standard Object Modeling Language*. Addison Wesley Professional, third edition, 2003.

[Gas02] Matthew S. Gast. *802.11 Wireless Networks: The Definitive Guide*. O'Reilly, first edition, 2002.

[GBRP03] Sumit Gwalani, Elizabeth M. Belding-Royer, and Charles E. Perkins. AODV-PA: AODV with path accumulation. In *IEEE International*

Conference on Communications (ICC' 03), volume 1, pages 527–531, Anchorage, Alaska, May 2003. IEEE.

[GKN+04] Robert S. Gray, David Kotz, Calvin Newport, Nikita Dubrovsky, Aaron Fiske, Jason Liu, Christopher Masone, Susan McGrath, and Yougu Yuan. Outdoor experimental comparison of four ad hoc routing algorithms. In *MSWiM '04: Proceedings of the 7th ACM international symposium on Modeling, analysis and simulation of wireless and mobile systems*, pages 220–229, New York, NY, USA, 2004. ACM Press.

[Glo06] Wolfram Gloger. Wolfram gloger's malloc homepage. `http://www.malloc.de/en/`, June 2006.

[GWW04] Abhinav Gupta, Ian Wormsbecker, and Carey Williamson. Experimental evaluation of TCP performance in multi-hop wireless ad hoc networks. In *Proceedings of the The IEEE Computer Society's 12th Annual International Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunications Systems (MASCOTS'04)*, pages 3–11, Washington, DC, USA, October 2004. IEEE.

[He05] Kevin He. Why and how to use netlink socket. *Linux Journal*, 2005(130):11, 2005.

[IdFC03] Roberto Ierusalimschy, Luiz Henrique de Figueiredo, and Waldemar Celes. *Lua 5.0 Reference Manual*. Tecgraf, PUC-Rio, 2003.

[Ier03] Roberto Ierusalimschy. *Programming in Lua*. Roberto Ierusalimschy, Rio de Janeiro, first edition, 2003.

[Ipw06] Intel® pro/wireless 2100 driver for linux. `http://ipw2100.sourceforge.net/`, September 2006.

[ITG] D-ITG, distributed internet traffic generator. `www.grid.unina.it/software/ITG`.

[JCH84] Rajandra K. Jain, Dah-Ming W. Chiu, and William R. Hawe. A quantitative measure of fairness and discrimination for resource allocation in shared computer systems. Technical Report TR-301, DEC, September 1984.

[JMH04] David B. Johnson, David A. Maltz, and Yih-Chun Hu. The dynamic source routing protocol for mobile ad hoc networks (DSR). Internet-Draft Version 10, IETF, July 2004. draft-ietf-manet-dsr-10, (Work in Progress).

[Jon] Rick Jones. Netperf homepage. `http://www.netperf.org/netperf/NetperfPage.html`. Last accessed November 2006.

[JS04] Glenn Judd and Peter Steenkiste. Repeatable and realistic wireless experimentation through physical emulation. *SIGCOMM Computer Communication Review*, 34(1):63–68, 2004.

[KAA06] A. Karygiannis, E. Antonakakis, and A. Apostolopoulos. Host-based network monitoring tools for MANETs. In *PE-WASUN '06: Proceedings of the 3rd ACM international workshop on Performance evaluation of wireless ad hoc, sensor and ubiquitous networks*, pages 153–157, New York, NY, USA, 2006. ACM Press.

[KBa] Luke Klein-Berndt. Kernel AODV. `http://w3.antd.nist.gov/wctg/aodv_kernel`.

[KBb] Luke Klein-Berndt. NIST dymo. `http://www-x.antd.nist.gov/twiki/bin/view/ANTDProjects/NistDymo`. Last accessed September 2006.

[KNG+04] David Kotz, Calvin Newport, Robert S. Gray, Jason Liu, Yougu Yuan, and Chip Elliott. Experimental evaluation of wireless simulation assumptions. In *MSWiM '04: Proceedings of the 7th ACM international symposium on Modeling, analysis and simulation of wireless and mobile systems*, pages 78–82, New York, NY, USA, 2004. ACM Press.

[KNSW02] Frank Kargl, Jürgen Nagler, Stefan Schlott, and Michael Weber. Ein framework für MANET routing protokolle. In *Proceedings of WMAN'02*, Ulm, Germany, March 2002. In German. An english version can be found at: `http://medien.informatik.uni-ulm.de/~frank/research/manetframework.pdf`.

[KP99] Brian W. Kernighan and Rob Pike. *The Practice of Programming*. Addison Wesley, 1999.

[KR88] Brian W. Kernighan and Dennis M. Ritchie. *The C Programming Language*. Prentice Hall, second edition, 1988.

[KR01] James T. Kaba and Douglas R. Raichle. Testbed on a desktop: Strategies and techniques to support multi-hop MANET routing protocol development. In *MobiHoc '01: Proceedings of the 2nd ACM international symposium on Mobile ad hoc networking & computing*, pages 164–172, New York, NY, USA, October 2001. ACM Press.

[KU03] Koojana Kuladinithi and A. Udugama. JAdhoc system design manual. Technical report, ComNets, IKOM, University of Bremen, July 2003.

[KUFG] Koojana Kuladinithi, Asanga Udugama, Nikolaus A. Fikouras, and Carmelita Görg. Experimental performance evaluation of AODV implementations in static environments. `http://www.comnets.uni-bremen.de/~koo/AODV-Perf-ComNets.pdf`.

[Kul05] Koojana Kuladinithi. MANET implementations. http://www.comnets.uni-bremen.de/~koo/manet-impl.html, January 2005.

[KZG03] Vikas Kawadia, Yongguang Zhang, and Binita Gupta. System services for ad-hoc routing: Architecture, implementation and experiences. In *Proceedings of the 1st International Conference on Mobile Systems, Applications, and Services (MobiSys)*, pages 99–112, San Francisco, CA, USA, May 2003.

[KZTM05] Wolfgang Kieß, Stephan Zalewski, Andreas Tarp, and Martin Mauve. Thoughts on mobile ad-hoc network testbeds. In *Proceedings of IEEE ICPS Workshop on Multi-hop Ad hoc Networks: from theory to reality*, pages 93–100, July 2005.

[Lea00] Doug Lea. A memory allocator. http://g.oswego.edu/dl/html/malloc.html, April 2000.

[Len05] Ricardo Lent. A testbed validation tool for MANET implementations. In *13th IEEE International Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems*, pages 381–388. IEEE, September 2005.

[Les] Antony Lesuisse. Airnet, wireless network. http://web.archive.org/web/20050308200036/http://airnet.org/. Original site is now unavailable. The link is a cached archive. Last accessed September 2006.

[LLN+01] Henrik Lundgren, David Lundberg, Johan Nielsen, Erik Nordström, and Christian Tschudin. A large-scale testbed for reproducible ad hoc protocol evaluations. Technical Report 2001-029, IT Department, Uppsala University, November 2001.

[LNT02] Henrik Lundgren, Erik Nordström, and Christian Tschudin. Coping with communication gray zones in IEEE 802.11b based ad hoc networks. In *WOWMOM '02: Proceedings of the 5th ACM international workshop on Wireless mobile multimedia*, pages 49–55, New York, NY, USA, 2002. ACM Press.

[Mal98] Gary Malkin. RIP version 2. RFC 2453, IETF, November 1998. rfc2453.txt.

[MANa] Mobile ad-hoc networks (manet) charter. http://www.ietf.org/html.charters/manet-charter.html. Last accessed November 2006.

[Manb] Ariel Manzur. tolua++. http://www.codenix.com/~tolua. Last accessed September 2006.

[MBJ99] David A. Maltz, Josh Broch, and David B. Johnson. Experiences designing and building a multi-hop wireless ad hoc network testbed. Technical Report CMU-CS-99-116, School of Computer Science, Carnegie Mellon University, Pittsburgh, Pennsylvania, March 1999.

[MBJ00] David A. Maltz, Josh Broch, and David B. Johnson. Quantitative lessons from a full-scale multi-hop wireless ad hoc network testbed. In *Proceedings of the IEEE Wireless Communications and Networking Conference*, volume 3, pages 992–997, Chicago, IL, USA, September 2000. IEEE.

[MBR+] James Morris, Marc Boucher, Rusty Russell, Harald Welte, Jozsef Kadlecsik, Martin Josefsson, Patrick McHardy, and Yasuyuki Kozakai. Netfilter - firewalling, NAT, and packet mangling for linux. `http://netfilter.org`. Last accessed September 2006.

[Moy98] John Moy. OSPF version 2. RFC 2328, IETF, April 1998. rfc2328.txt.

[MS04] Neil Matthew and Richard Stones. *Beginning Linux Programming*. Wiley, third edition, 2004.

[Net99] Netlink(7), April 1999. Linux Programmer's Manual.

[NGL05] Erik Nordström, Per Gunningberg, and Henrik Lundgren. A testbed and methodology for experimental evaluation of wireless mobile ad hoc networks. In *Proceedings of the First International Conference on Testbeds and Research Infrastructures for the DEvelopment of NeTworks and COMmunities (TRIDENTCOM'05)*, pages 100–109, Washington, DC, USA, 2005. IEEE.

[Nie] Gustavo Niemeyer. Lunatic-Python. `http://labix.org/lunatic-python`. Last accessed September 2006.

[Nor] Erik Nordström. AODV-UU. `http://core.it.uu.se/core/index.php/AODV-UU`. Last accessed December 2006.

[OPN] OPNET. `http://www.opnet.com`. Last accessed September 2006.

[OTL04] Richard G. Ogier, Fred L. Templin, and Mark G. Lewis. Topology dissemination based on reverse-path forwarding (TBRPF). RFC 3684, IETF, February 2004. rfc3684.txt.

[Pal06] Mike Pall. The LuaJIT project. `http://luajit.luaforge.net/index.html`, 2006.

[PB94] Charles E. Perkins and Pravin Bhagwat. Highly dynamic destination-sequenced distance-vector routing (DSDV) for mobile computers. In *SIGCOMM '94: Proceedings of the conference on Communications architectures, protocols and applications*, pages 234–244, New York, NY, USA, 1994. ACM Press.

[PBRD03] Charles E. Perkins, Elizabeth M. Belding-Royer, and Samir R. Das. Ad hoc on-demand distance vector (AODV) routing. RFC 3561, IETF, July 2003. rfc356.txt.

[PBRDC] Charles Perkins, Elizabeth Belding-Royer, Samir Das, and Ian Chakeres. AODV homepage. `http://moment.cs.ucsb.edu/AODV/aodv.html#DYMO`.

[Pro] Kepler Project. LuaJava, a script tool for java. `http://www.keplerproject.org/luajava/`. Last accessed September 2006.

[RABR05] Krishna N. Ramachandran, Kevin C. Almeroth, and Elizabeth M. Belding-Royer. A framework for the management of large-scale wireless network testbeds. In *Proceedings of the 1st workshop on Wireless Networks Measurements (WinMee)*, April 2005.

[RGK+04] Miguel Rio, Mathieu Goutelle, Tom Kelly, Richard Hughes-Jones, Jean-Philippe Martin-Flatin, and Yee-Ting Li. A map of the networking code in linux kernel 2.4.20. Technical Report DataTAG-2004-1, FP5/IST DataTAG Project, Research & Technological Development for a TransAtlantic Grid, March 2004.

[RP00] Elizabeth M. Royer and Charles E. Perkins. An implementation study of the AODV routing protocol. In *Proceedings of the IEEE Wireless Communications and Networking Conference*, volume 3, pages 1003–1008, Chicago, IL, USA, September 2000. IEEE.

[RR] Fransico J. Ros and Pedro M. Ruiz. DYMOUM. `http://masimum.dif.um.es/?Software:DYMOUM`. Last accessed December 2006.

[Rus] Rusty Russell. Linux netfilter hacking howto: Information for programmers. `http://netfilter.org/documentation/HOWTO/netfilter-hacking-HOWTO.html`. Last accessed September 2006.

[SFR03] W. Richard Stevens, Bill Fenner, and Andrew M. Rudoff. *The Sockets Networking API*, volume 1 of *UNIX Network Programming*. Addison Wesley Professional, third edition, 2003.

[Sho] The computer language shootout benchmarks. http://shootout.alioth.debian.org. Last accessed September 2006.

[TO04] Christian Tschudin and Evgeny Osipov. Estimating the ad hoc horizon for TCP over IEEE 802.11 networks. In *Proceedings of the 3rd Annual Mediterranean Ad Hoc Networking Work- shop, Med-Hoc-Net*, pages 255–262, Bodrum, Turkey, June 2004.

[Tou04] Jean Tourrilhes. The devices, the drivers. http://www.hpl.hp.com/personal/Jean_Tourrilhes/Linux/Linux.Wireless.drivers.802.11b.html, August 2004.

[TQD+05] Ajay Tirumala, Feng Qin, Jon Dugan, Jim Ferguson, and Kevin Gibbs. Iperf – the TCP/UDP bandwith measurement tool. http://dast.nlanr.net/Projects/Iperf/, May 2005.

[Wes03] David West. An implementation and evaluation of the ad-hoc on-demand distance vector routing protocol for windows CE. Master's thesis, University of Dublin, September 2003.

[Wib02] Björn Wiberg. Porting AODV-UU implementation to ns-2 and enabling trace-based simulation. Master's thesis, Uppsala University, December 2002.

[ZL02] Yongguang Zhang and Wei Li. An integrated environment for testing mobile ad-hoc networks. In *MobiHoc '02: Proceedings of the 3rd ACM international symposium on Mobile ad hoc networking & computing*, pages 104–111, New York, NY, USA, 2002. ACM Press.